Dr. Dobb's Journal

September 2013

Distributed Version Control

Understanding the Paradigm Shift

Next

ALSO INSIDE

UBM Tech

www.drdobbs.com

Dr. Dobb's Journal

CONTENTS

September 2013

COVER ARTICLE

FEATURES

More on DrDobbs.com

Developer Reading List

New books on Windows internals, JavaScript, Groovy, Python, and P=NP.

http://www.drdobbs.com/240160157

Overcoming HTML5's Limitations

HMTL5 is such a low-cost and portable alternative to native app development that it makes sense to explore solutions that address its limitations.

http://www.drdobbs.com/240159696

Atomic Operations and Low-Wait Algorithms in CUDA

Used correctly, atomic operations can help implement a wide range of generic data structures and algorithms in the massively threaded GPU programming environment. However, incorrect usage can turn massively parallel GPUs into poorly performing sequential processors.

http://www.drdobbs.com/240160177

Building Web Apps with Lift: Lazy Loading, Push, and Parallel Execution

Lift code is easy to read and maintain. Lazy loading, parallel execution, simple push mechanics, and REST support are just a few of Lift's stand-out benefits.

http://www.drdobbs.com/240159513

Dr. Dobb's Now Has A Windows Phone App

A reader writes an open-source app for Windows Phone 8.

http://www.drdobbs.com/240159570

Mailbag

Advice to a New Programmer

In response to our editorial on five foundational practices for new programmers (http://www.drdobbs.com/240158341), we received these thoughtful letters:

"I loved your recent article 'Advice to a New Programmer.' I've been programming in C++ for 16 years now, but I feel like a new programmer. My organization has terrible and naïve code since we've rarely had people on the team who are good at software quality, so I still consider myself a pretty lousy programmer. Getting advice from an accomplished software professional like you is music to my ears! I loved your suggestions. I plan to seek out some really good C++ code and books to read, and just in general to write more code (my role is now more of a tech lead than a coder).

Thank you for your article. I would love to see a follow up some day!"

— [name withheld]

"I read your editorial, 'Advice to A New Programmer,' and decided to get in touch. I am trying to move into Web development, but don't know whether to study PHP or JavaScript first.

I hope you will tell me the path to follow.

— M. Kokalov
Sofia, Bulgaria

Andrew Binstock responds: "Both languages are comparatively easy to learn, although PHP is probably easier. However, JavaScript is becoming a universal language for Web, mobile, and desktop apps. So I would strongly favor JavaScript."

Oldy But Goody

"I just came across your editorial 'Will Parallel Code Ever be Embraced?' (http://www.drdobbs.com/240003926), which you wrote about a year ago. I enjoyed reading it.

Ever since multicore CPUs have been available, I have been trying to figure out how developers are going to effectively make use of. As you know, the CPU vendors didn't develop multicore chips because they thought users wanted multiple cores; they did it because they couldn't keep increasing the clock speeds of single-core CPUs without creating little furnaces inside computers.

In the embedded computing industry, customers have been using multiple processors for years — distributed multi-processing — as you put it, the "embarrassingly parallel" applications like signal and image processing (e.g., radar). They have done this because no single processor was powerful enough to run their algorithms, so they were forced to distribute the problem among multiple CPUs to get real-time performance.

When a CPU has 2 or 4 cores, a smart engineer can probably figure out how he or she can distribute an application across the cores to (more or less)effectively take advantage of the cores. But, what about when there are 8 cores, 16 cores, 24 cores, or more. I think developers are going to need a lot of help from their development tools.

What about the development tools? Are there development tools that make it easier to distribute an application across multiple cores? I keep asking people, but I haven't received an answer that satisfied me. And how do developers effectively make use of multicore chips?

— Dave Barker

Andrew Binstock responds: "For traditional threading (fine-grained), the most advanced set of tools is made available by Intel. They have thread checkers, profilers, and other tools. They also have the Cilk+ framework to facilitate low-level coding, as well as Threading Building Blocks (TBB). Point products such as Rogue Wave's TotalView debugger are also highly regarded.

As to coarser-grained threading, the emerging model seems to favor message passing. Languages are increasingly adding actor-like, message-passing capabilities into the language itself (Scala and Erlang) or into the libraries (Groovy, Java, and several others) to attain parallel execution."

Have a correction or a thoughtful opinion on *Dr. Dobb's* content? Let us know! Write to Andrew Binstock at alb@drdobbs.com. Letters chosen for publication may be edited for clarity and brevity. All letters become property of *Dr. Dobb's*.

News Briefs

By Adrian Bridgwater

Real Agile Means Everybody Is Agile

Agile project management company VersionOne has added new functions named PlanningRoom, Conversations+ ("Conversations Plus"), and SAFe Metrics as part of its summer 2013 Release. PlanningRoom is a program-level planner that can be used in review meetings and works as a role-based portal that feeds into all other areas of the application; it is capable of planning and tracking capabilities that support complex program hierarchies. SAFe Metrics reports works to help measure the progress of a customer's enterprise Agile initiatives using the Scaled Agile Framework (SAFe). Users can get data regarding asset health, release predictability, feature completeness, and program performance.

http://www.drdobbs.com/240159622

NoFlo Champions Flow-Based Programming

Flow-based software development company NoFlo is bidding to transform software development from a text-oriented process to a visual object-oriented process. The firm wants developers to be able to see what is happening as a result of code deployed and make changes while a component is running. NoFlo is based on "Flow-Based Programming" — with its roots at IBM in the 1970s, flow-based programming has been widely used for the creation of 3D and special effects in movies. Everything in the software development process is always a visual graph. So not just visual for the original design, but every single component is visually connected and can be reordered and reused.

http://www.drdobbs.com/240159610

New Eclipse Dawns On Machine-2-Machine

The Eclipse Foundation's M2M Working Group has added new Machine-2-Machine (M2M) and Internet of Things (IoT) development projects to its current cadre of interests. M2M connection between machines and devices (via a network) is one of the fastest growing technology segments — by some estimates approaching $100 billion in total market value. This new technology segment presents a number of software development challenges due to the complexity of the hardware and network architectures, lack of open standards, and issues of interoperability between vertical solutions. The Eclipse M2M Working Group is a collaboration of companies working to address this lack open protocols, frameworks, and tools for M2M software development. Eclipse M2M has three existing open source projects — Koneki, Mihini, and Paho — that provide open source technology for M2M developers.

http://www.drdobbs.com/240159487

Engine Yard Burns Azure Blue

The journey to multi-cloud cross-platform programming Nirvana may be one step closer with the release of Engine Yard on Windows Azure. Developers can now use this open source Platform-as-a-Service running on Microsoft's cloud infrastructure to deploy web and mobile apps. Windows Azure customers can purchase an Ubuntu-based, predefined configuration, optimized by Engine Yard, that manages five servers. This offering is a basic add-on in the Windows Azure Store and includes two servers in an HAProxy load balancing cluster, two servers in a Ruby application cluster, and one server in a MySQL Percona cluster.As part of its support for Windows Azure, Engine Yard is also introducing a new user interface, based on AngularJS.

http://www.drdobbs.com/240159659

Distributed VCS: Understanding the Paradigm Shift

Distributed Version Control Systems use a different model of check-out, check-in, branching, and merging than traditional, centralized, repository-based solutions. Understanding these differences is the only way to enjoy the benefits of distributed version control.

By Pablo Santos

Distributed Version Control Systems (DVCSs) are here to stay, steadily replacing the former generation of centralized version control systems. What are the key concepts to keep in mind as an SCM manager or senior developer to transition from your current system to the new generation? Is it just about the "multi-site" capabilities? Do you need to structure your repositories differently? How is it going to affect your day-to-day operations? Do you need a different mindset?

My goal with this article is to answer these questions and describe how to embrace and get the best out of the new generation of DVCSs.

Not Just About Its Distributed Nature

When we think about DVCSs, the first thing that comes to mind is their ability to work disconnected from a central server. A DVCS is about having a local repository on your machine, from which you can check-in code, branch, and merge locally. You then can push your changes to another server (a central server, a peer system, or whatever configuration you're working on).

The basic operations working in DVCS are those illustrated in Figure 1.

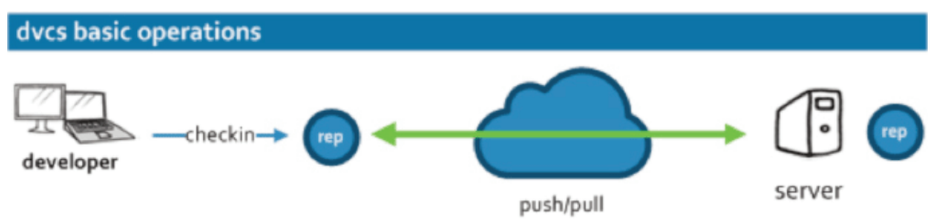


Figure 1: Overview of basic check-in and push in a DVCS.

The basic approach is:

- You always work against a local repository
- Check-ins go to the local repository (on your own machine)
- Then you "push" the changes to the remote server
- Or "pull" new changes from the remote server

But, as the title says, DVCS is not just about the "D" in distributed. Distribution is the feature getting the flashy headlines, but the key points behind the paradigm shift are the ones I'm about to describe. The new DVCSs come with several important features, which I'll explain in this article:

- Whole repository versus file revisions
- Consistent and immutable configurations
- Vastly improved merge tracking — not only from the functional standpoint, but also performance wise
- Ability to implement well-known (though commonly feared) branching patterns

A File-by-File World View

I used to imagine repository evolution in terms of files evolving separately. Several well-known version control systems taught us to think in the following way: Files evolve separately, with version numbers on their own, and we had the tools to configure them in different ways… even dynamically.

Figure 2 is an illustration of the standard way of looking at file changes might look familiar: The four featured files have different revisions and we can easily check their individual history, diff their

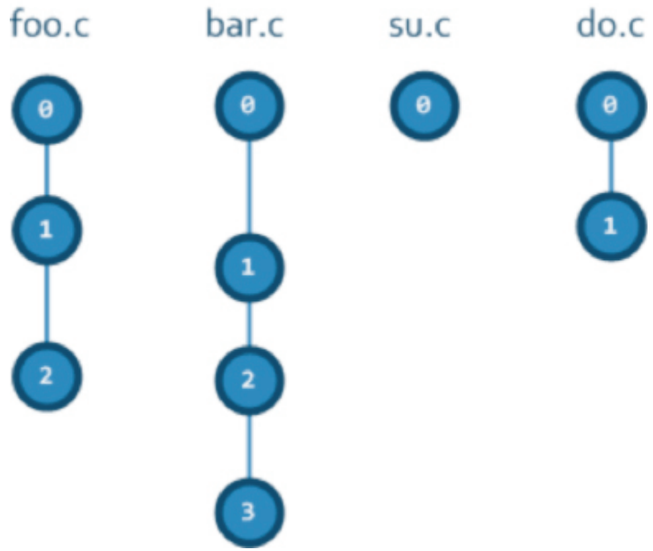


Figure 2: Standard way of looking at file changes.

different revisions, and also download them to our working areas, which I'll refer to as workspaces from now on (but are also known as views, working copies, and all sort of different names that version control creators use to confuse developers).

At this point, labels come to the rescue: You can group revisions using a label or tag, which is a means to retrieve the file versions as a group at a later point. So, we can create a "version 1.0" label to mark the revision 1 of foo.c, bar.c rev 2, su.c rev 0, and doc.c rev 1, as Figure 3 shows.

This option enables us to take a snapshot of the entire structure from time to time to be able to create solid points that we can recover later on and use as baselines, checkpoints, or whatever we want to call them.

This file-based approach is great from a power developer point of view because you can do great things such as download to your workspace the revision 0 of foo.c, bar.c rev 3, su.c rev 0, and do.c rev 0, and work with them. But this is flexibility that comes at a cost: What exactly has been downloaded? Did this configuration ever exist as such? Maybe it was never "this way" on any developer workspace, so there's no way the files will work together correctly. Should we even bother with this sort of dynamic configuration then? It is a matter of taste, so I won't be entering this discussion now.

One Revision To Rule Them All

In the DVCS world view, things happen slightly differently. Instead of looking at your history file by file, you get used to checking what happened at a repository level. A key concept is the changeset (or commit), which is not unique to DVCS, but plays a different role. A changeset is a set of files that are checked-in together. A check-in is similar to a database transaction: If you want to check-in four files together, a new

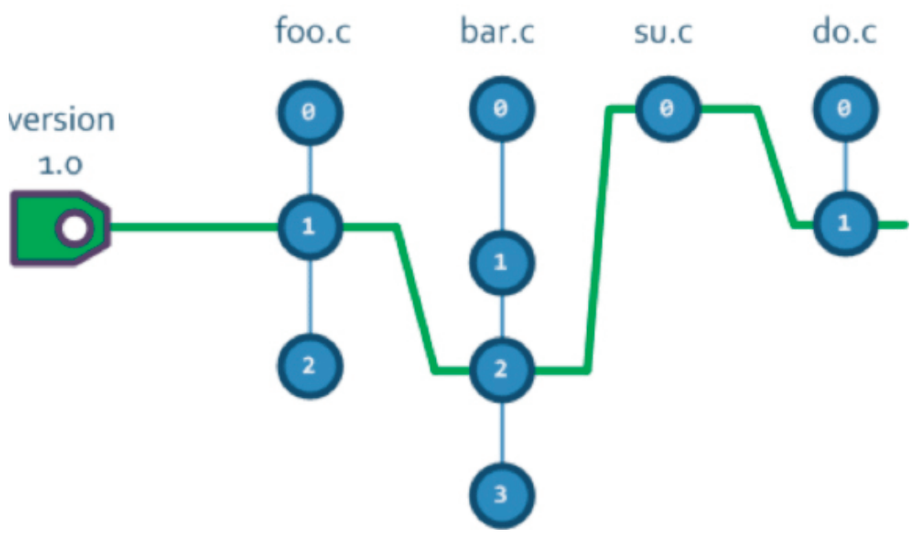


Figure 3: Using tags or labels to group multiple file revisions.

changeset grouping the four files will be created only if the check-in succeeds for all four files. Otherwise, the transaction will be rolled back and no new changeset is created. In other words, a changeset is the result of an atomic check-in.

In DVCS, check-ins become first-level players; in fact, they become the centerpiece of the history of a repository. Let's go to our previous example: You create foo.c and bar.c and check them in. Something similar to what is shown in Figure 4 will happen.

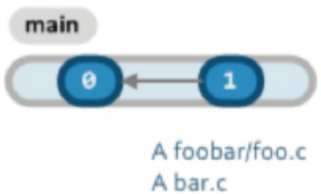


Figure 4: A simple check-in.

Suppose "changeset 0" is empty (the root empty changeset); then we've just created changeset 1, containing the two new files we added.

Let's continue and add su.c and do.c, and also make a change on foo.c (Figure 5).

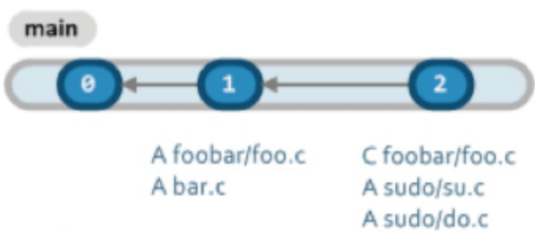


Figure 5: Adding more files.

Now, make another change to foo.c by moving bar.c to foobar/bar.c and also modifying it. Finally, modify do.c, too. Then make another change on bar.c (Figure 6).

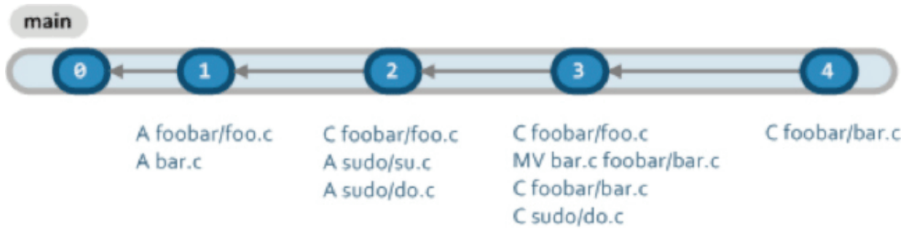


Figure 6: Even more changes.

As you can see, we've created exactly the same number of file revisions we did in the "file-by-file history" example discussed earlier, but we're not thinking about file history anymore; rather, we are tracking what happened to our project — to our code base as a whole.

The changeset is playing two roles:

- It captures the change made at each step: In changeset 3, I modified 3 files and I also made a move
- It captures a full snapshot of the project at each step, check-in after check-in (and this is one of the key points to keep in mind when moving to DVCS)

What is the purpose of the snapshot? Suppose you want to go back to changeset number 1: If you "download" changeset 1, your workspace will look like Figure 7.

Figure 7: Workspace after downloading changeset 1.

Now, if you go to changeset 3, it will look like Figure 8.

Figure 8: Workspace after downloading changeset 3.

And you can even diff the trees at any point and understand what happened between them. For instance, a diff between cset 3 and 1 will tell us:

- bar.c has been moved to foobar/bar.c
- sudo/su.c and sudo/do.c have been added
- foo.c and bar.c were also modified

The key concept to keep in mind is that each changeset is actually capturing the status of the project at a certain point. When a developer checked-in changeset 3, he had a certain configuration, a certain "tree," and this configuration is recorded in the repository and can be easily recovered.

In addition, every team I know enforces the rule: Check that the code builds correctly before check-in (even pass all tests) so you'll make sure every "snapshot" you're taking will effectively record a real configuration — something that really existed this way and could be built.

You'll notice that I'm stressing the concept of consistent check-ins versus dynamic configurations (where you download separate revisions of files to setup your workspace). It is a simplification of the previous model, and at first, you'll probably feel like you're losing power. But you're trading complexity for performance, simplicity and consistency (and better merging, which will be discussed shortly).

By the way, labels (or tags) are much easier to understand: Instead of a set that groups together random revisions, a tag is just an alias for a changeset (Figure 9).

The benefit is clear: The tag is easier to understand and much faster to apply (in some old version control systems, applying a label to a big codebase took ages because it involved a new entry to be set for each revision to be labeled. Now it is just a new entry, independent of your project files).

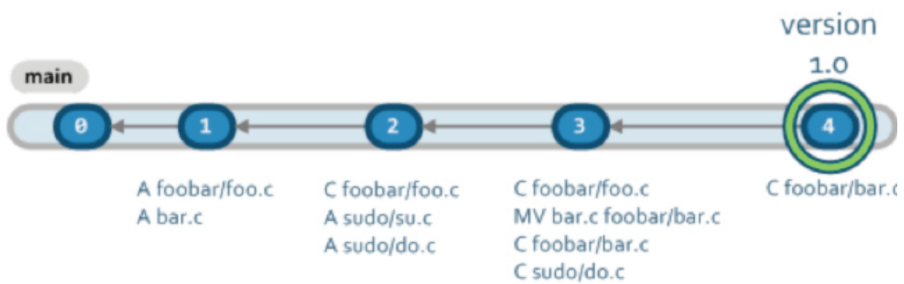


Figure 9: Applying a tag ("version 1.0") to a changeset (4).

Things That Have Changed

Concerning the changes I've described so far, Table 1 shows the things you do and have in a DVCS and things you don't.

You do or you have	You won't
Check the evolution of a project on a changeset-by-changeset basis. You'll check what files changed between two changesets	Look at the history of a file so often since you'll get used to thinking on a global basis.
Set your workspace to work on a given changeset, downloading a full configuration	Hand pick the files you will download to your workspace one-by-one or based on dynamic rules
Fast labeling	Wait for a label to be set

Table 1: The things you have in a DVCS and no longer use from traditional SCMS.

The Flexible Checkout Model vs. The Consistent and Rock Solid One

When I say "checkout," I mean it in the Subversion sense of checkout: Download files to your workspace (different from the "lock" or the "create new revision" that is performed in other systems). File-based version control systems allowed us to do things like that shown in Figure 10.

You just download part of the repository to work on it. While this is doable in DVCS, it is not normally the way in which things are done. In a DVCS, you don't follow the "monster-repo" model where you work just on subsets of the project. This means that you will not use the traditional model (Figure 11) for multiple projects.

In the traditional model, you define a big repository with all the code, then each developer must configure his/her working copy to work on the right parts of the code for a given project. The drawback is that the configuration of each workspace is dynamic and error-prone. Also, if you remember the "snapshot" approach I described earlier where each changeset is meant to capture a complete configuration, you'll see that

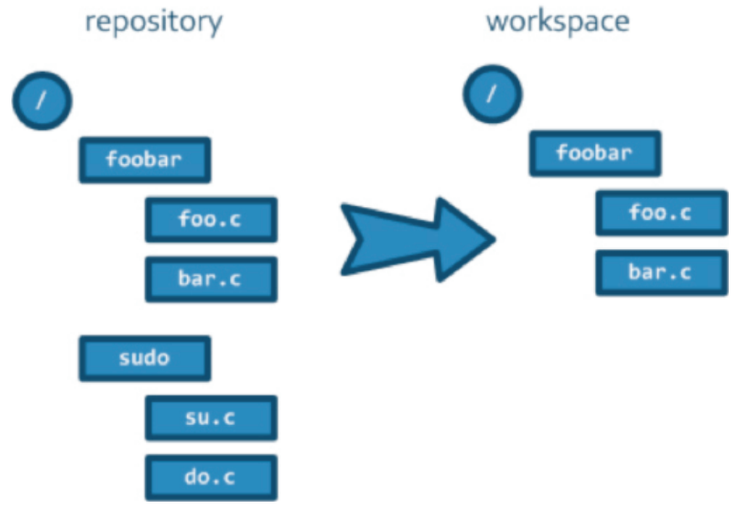


Figure 10: Checking out part of the repository.

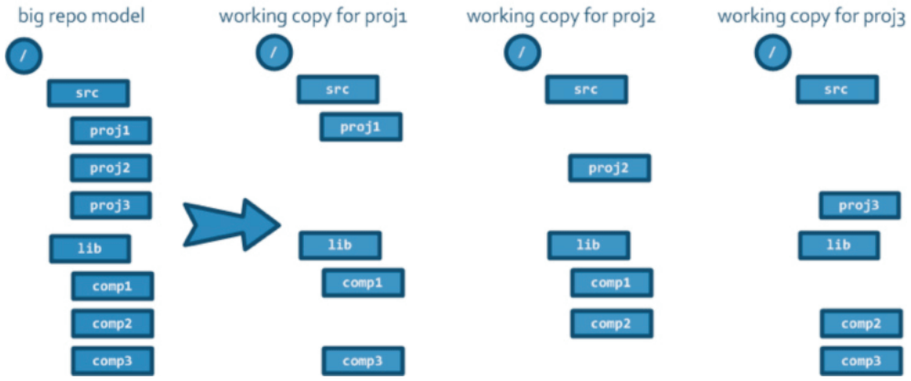


Figure 11: Working on multiple projects from the same repo in the traditional SCM model.

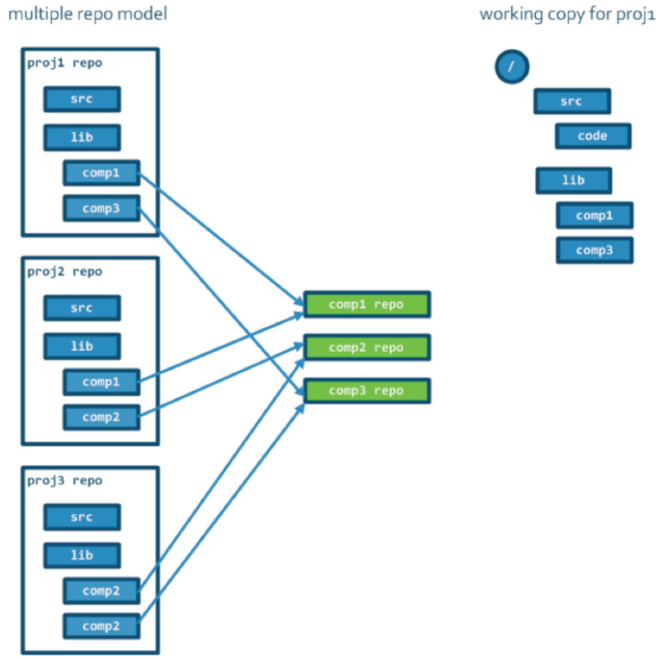


Figure 12: The multiple projects from Figure 11 in a DVCS.

the "traditional" way of working is not truly compatible. When a developer working on "proj1" checks-in, he will be creating a new changeset where there will be contents (src/proj2, src/proj3, lib/comp2) that are not related to his changeset and will be incorporated in the "snapshot" together.

In a pure DVCS approach, things would be a little bit different, as shown in Figure 12.

As you can see, some code reorganization is required, but the benefit is that every project directly maps to a repository in a one-to-one relationship. Every project is contained in a single repository, so the consistency of the changeset model is preserved. Also, configuring a workspace is as simple as downloading from the corresponding repository; there's no need for the developer to play with download rules or to decide what he needs. He has the whole repository. Gone is the error-prone part because the configuration is captured by the repository.

Multiple, consistent, and well-defined repositories are preferred, and the reason is deeply tied to two concepts:

- Each changeset captures a fully consistent picture of the status of the project at a given time
- Merge tracking in DVCSs is handled at the changeset level and not at file level. I'll cover this topic in the next section.

Merge Tracking in DVCS

The version control systems born before the age of DVCS implemented "item-based merge tracking," which means every single file had its own merge tracking history. Figure 13 shows the history of a single file in a system implementing item-based merge tracking.

The green arrow is a merge link and the graphic shows how the file "foo.c" evolved from the "main" branch. Next, a "branch3" was created, two new revisions were checked in there, then they were merged back

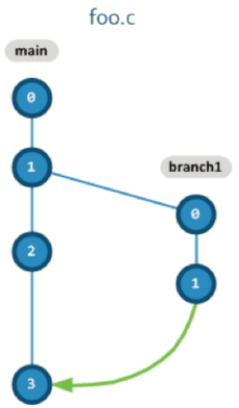


Figure 13: Item-based branch and merge in traditional SCM systems.

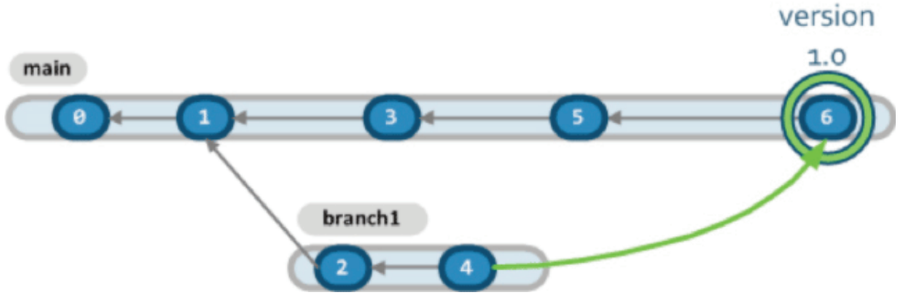


Figure 14: Branch and merge in changeset-based DVCSs.

to the "main" branch. In this model, every file and directory has its own history tree with merge history.

Figure 14 shows how things happen in a changeset-based merge tracking system, such as those used in DVCSs systems:

The main difference is that the merge tracking is not handled on an item-by-item basis (file-by-file or directory-by-directory), but on a global basis: The system keeps track of the merges on a changeset level. What are the important differences between the two models?

Changeset-based merge tracking enables much faster merging. Suppose you have to merge two branches: "main" and "feature130." The "feature130" branch was created from "main," but then the two branches evolved in parallel for a month. Major changes were done in "feature130," so a total of 20,000 files were modified there. In an item-based merge tracking system, the version control system will need to check the merge history of at least 20,000 items. You can optimize the merge code as much as you want, but it will never beat the speed of changeset-based merge tracking. The latter will check only one merge tracking tree — the changeset tree, which is going to be only one — independent of the number of files in the repository. This is really a significant advantage.

Changeset-based merge systems cannot do "partial merging." This is an important limitation that needs to be correctly understood. In practice, it means that when you merge a branch you will have to merge all the files or none of them. It is not possible (as it was with item-based merge tracking systems) to merge only a few files and then merge the rest of them later on. Merge tracking information is attached to the changeset, not kept on a file-by-file basis, and that's why merges can't be partial. At the end of the day, it doesn't impose such a big restriction because, conceptually, you won't do changes on branches that you won't integrate, and branches are handled as a unit of change that are merged as a complete unit.

This difference in merge tracking is one of the reasons why multiple cohesive repositories are preferred in DVCS over a single one with sparse checkouts.

Merging the Full Changeset or Nothing

Let's see why changeset-based merge tracking systems need to stick to the "merge the entire changeset or merge nothing" rule. Consider the events shown in Figure 15.

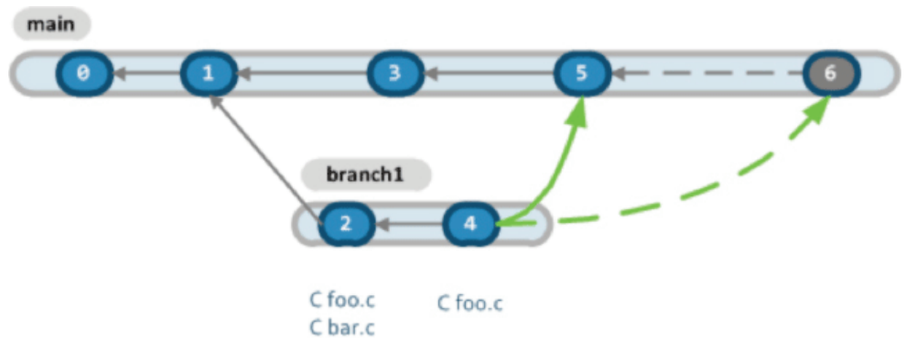


Figure 15: Merging in a DVCS.

In branch3, we modified files foo.c and bar.c. What happens if, during the merge from changeset "4" to "3" (creating changeset "5" as result), we only merge foo.c? In an item-based merge, the history of foo.c will include a new merge link, while bar.c will still not be merged. If we repeat the merge again to create changeset "6," then "bar.c" will be merged. In a DVCS, you cannot do this partial merge that forgoes bar.c.

The theory behind this way of implementing merge tracking states:

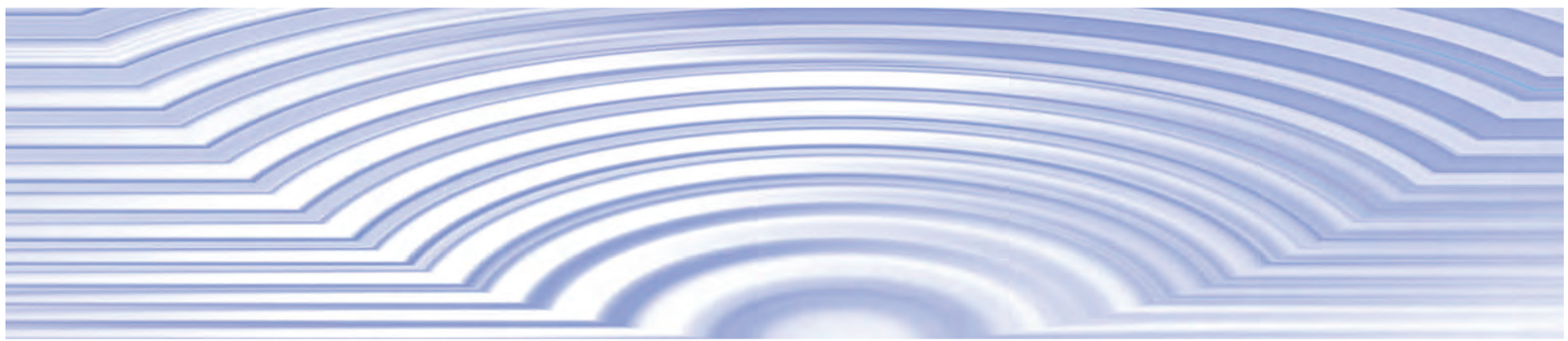
- If you don't merge certain files from a branch, most likely it means you're abandoning them. What else would force you to create a changeset "5" where part of "branch3" is not merged? By doing that, you wouldn't be preserving the consistency rule.
- If you use consistent configurations (a directory tree belongs to the same logic unit), you'll always be able to run complete merges. For instance: If you have a source tree with subdirectories "/src/tetris-game" and "/src/arcanoid-game," chances are you'll need to merge only the part of the "high-res-branch" related to "tetris-game" into the "tetris-2" branch. But, following this train of thought, shouldn't "src/tetris-game" and "src/arcanoid-game" belong to different repositories? In the DVCS world, the answer would be "yes" and this way, you avoid the need for most partial merges.

Conclusion

DVCSs present a new approach to version control that is constrained to their "distributed" nature, and also introduce changes in the way in which repositories are designed and structured. Getting used to this new DVCS approach is the key to getting the best out of the next generation of version control systems and correctly structuring your projects.

— Pablo Santos is the founder and chief engineer of Codice Software, the makers of PlasticSCM.

Comment



Getting Started with Git: The Fundamentals

The distributed SCM system that's taking the world by storm has its own unique way of doing things. This tutorial explains how things work and the basic commands for getting started and checking-in changes

By Scott Danzig

I was not particularly inspired by any SCM system until I dove into Git, created by Linus Torvalds, the founder of Linux. In this tutorial, I discuss what's unique about Git and I demonstrate how to set up a repository on GitHub, one the main free Git hosting services. Then I explain how to make a local copy of the GitHub repository, make some changes locally, and push them back to GitHub. The second part of this tutorial (*Git Tutorial: Branches and Workflow* on page 27) builds on this base, explains branching and merging, and discusses a workflow that I use, which might be of interest to you. As a side note, I learned much of what I know about Git from the book *Pro Git*, which is hosted free online at http://git-scm.com/book. I recommend that you use the book to fill out the matter presented here and as a reference for later work with Git.

Why Git?

Git has numerous attractive benefits that, for me, make it my preferred DVCS:

- When you create a new branch, Git doesn't copy all your files over. A branch will point to the original files and only track the changes (commits) specific to that branch. This makes it blazingly fast to create branches compared to other approaches, such as Subversion (which laboriously copies the files).
- Git lets you work on your own copy of a project, merging your commits into the central repository, often on GitHub.com, when you want your commits to be available to others. Github.com, by the way, will host your project for free as long as it's open source. (And cheaply, if it's not. Another alternative is Bitbucket, which allows unlimited private Git repositories.) This means you can reliably access your code from anywhere with an Internet connection. If you lose that Internet connection, you can continue to work locally and sync up your changes when you're able to reconnect.
- When you screw up, you can usually undo your changes. You might need to call in an expert in serious cases, but there's always hope.

This is the best "key benefit" a version control system can have.

- Git also lets you keep your commit history very organized. If you have lots of little changes, it lets you easily rewrite history so you see it as one big change (via something called "rebasing"). You can add/remove files in each commit, and certainly change the descriptions of each.
- It's open source, fast, and very flexible, so it's widely adopted and well-supported.
- With Git, you can create "hooks," which enable actions to occur automatically when you work on your code. A common use case is to create a hook to check the description submitted with each commit to make sure it conforms to a particular format. Perhaps you have your bugs described in a bug tracking system, and each bug has an ID #. Git can ensure each message has an entry for "Bug: SomeNumber".
- Another under-appreciated feature is how Git tracks files. It uses the SHA-1 algorithm to take the contents of files and produce a large hexadecimal number (hash code). The same file will always produce the same hash code. This way, if you move a file to a different folder, Git can detect that the file moved, and not think that you deleted one file and added another. This allows Git to avoid keeping two copies of the same file.
- While Git is not necessarily the most intuitive version control system out there, once you get used to it, you're able to browse through its internal directories and it makes complete sense. Wondering where the file with the hash code "`d482acb1302c49af36d5dabe0bccea04546496f7`" is? Check out this file: "`<your project>/.git/objects/d4/82acb1302c49af36d5dabe0bccea04546496f7`"

There are also lots of lower-level commands that let you build the operations you want, in case, for instance, Git's `merge` command doesn't work how you'd like it to.

Tutorial

Let's jump in. In whatever programming language, you're going to start a new project, and you want to use version control. I'm going to create a silly, sample application in Scala that's very easy to understand for a demonstration. I'll assume you're familiar with your operating system's command-line interface, and that you're able to write something in the language of your choice.

Setup

Github is one of the go-to places to get your code hosted for free and it's what I'll use here. (BitBucket, Google Code, and SourceForge are some of the other free repository hosts that support Git). All these hosts give you a home for your code that you can access from anywhere. Initial steps:

- Go to http://GitHub.com and "Sign up for Github"
- You'll need Git. Follow this step-by-step installation process
- Review how to create a new repository
- Finally, you're going to want to get used to viewing files that start with a "." These files are hidden by default; so at the command line, when you're listing the contents of a directory, you need to include an "`a`" option. That's "`ls -a`" in OS X and Linux, and "`dir /a`" for Windows. In your folder options, you can turn on "Show hidden files and folders" as well.

Once you get this far, there's nothing stopping you (outside of setting aside some time to explore what Git has to offer). Let's look at some of the typical actions.

Clone a Repository

Cloning a repository lets you grab the source code from an existing project (yours or someone else's) that you have access to. Unless it's your project, you won't be able to make changes unless you "fork" the project, which means creating your own copy of it under your own account, after which you can modify it to your heart's content. I keep all of my projects locally (on my computer) in a "projects" folder in my home directory, "/Users/sdanzig/projects", so I'm going to use "projects" for this demo.

First, I fork my repository…

I create a sample project, called potayto, on GitHub, as you now should know how to do. Let's get this project onto your hard drive so you can add comments to my source code for me. First, log into your GitHub account, then go to my repository at https://GitHub.com/sdanzig/potayto and click Fork, see Figure 1.

Then select your user account on GitHub and copy it there. When this is complete, it's as though it were your own repository and you can actually make changes to the code on GitHub. Now, let's copy the repository onto your local hard drive, so we can both edit and compile the code there, see Figure 2.

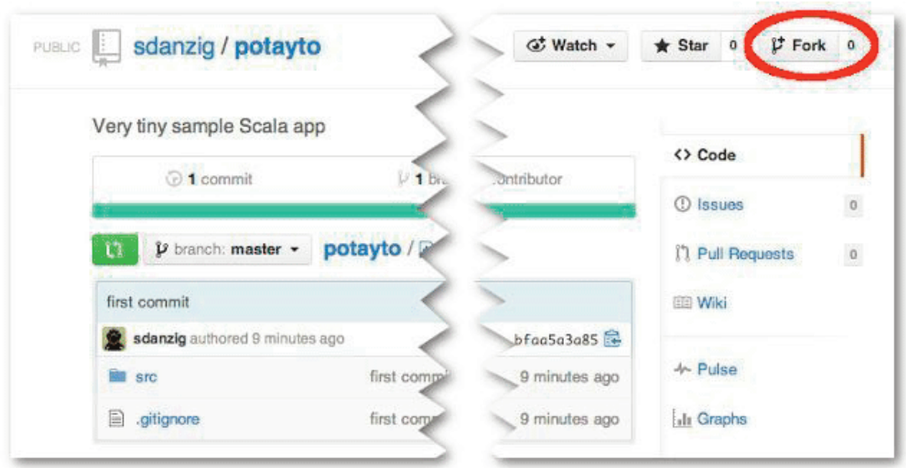


Figure 1: Forking (cloning) a repository.

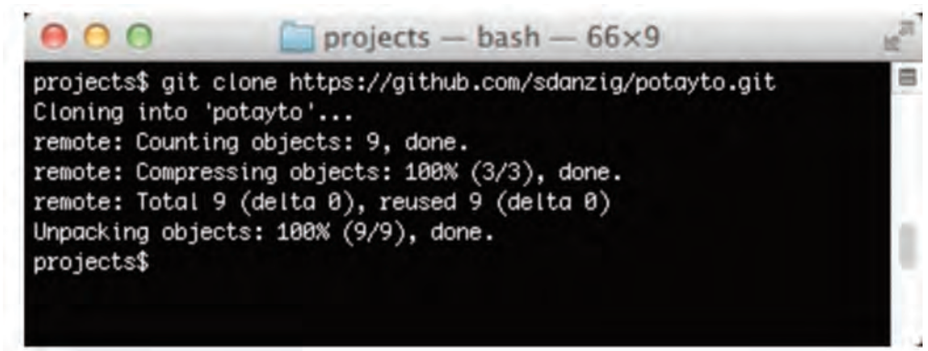


Figure 2: Copying a repository.

Folder Structure

There are a few key things to know about what Git is doing with your files. Type: `cd potayto`. There are useful things to see here when you list the contents in the potayto folder, being careful to show the hidden files and folders (with the `-a` option), see Figure 3.

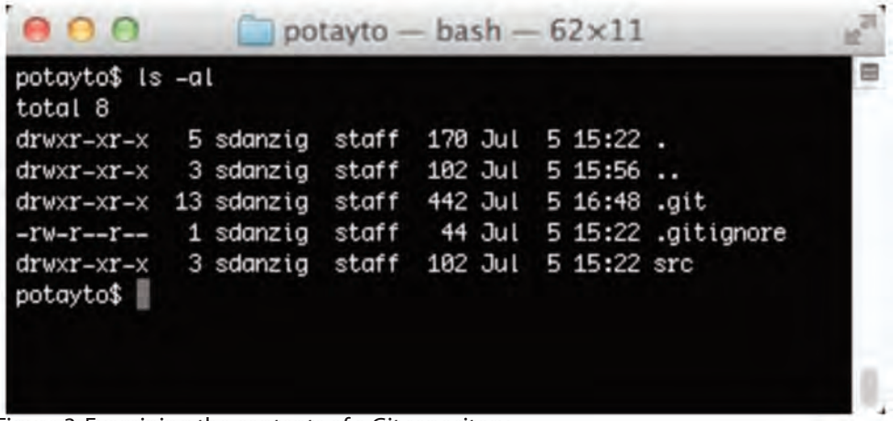


Figure 3: Examining the contents of a Git repository.

The src folder contains the source code, and its structure conforms to the Maven standard directory structure. You'll also see a .git folder, which contains a complete record of all the changes that were made to the potayto project, as well as a `.gitignore` text file. We're not going to dive into the contents of .git in this tutorial, but it's easier to understand than you think. If you're curious, please refer to the free online book.

Git Log

A "commit" is a change recorded in your local repository. Type "`git log`," and you might have to press your space bar to scroll and type "`q`" at the end to quit displaying the file, see Figure 4.

Git's log shows that the potayto project has 3 commits so far, from the oldest on the bottom to the most recent on top. You see the big hexadecimal numbers preceded by the word "commit"? Those are the SHA-1 codes I referred to earlier. Git also uses these SHA-1 codes to identify commits. They're big and scary, but you can just copy and paste them. Also, you need to type only enough letters and numbers for it to be uniquely identified (five is usually enough).

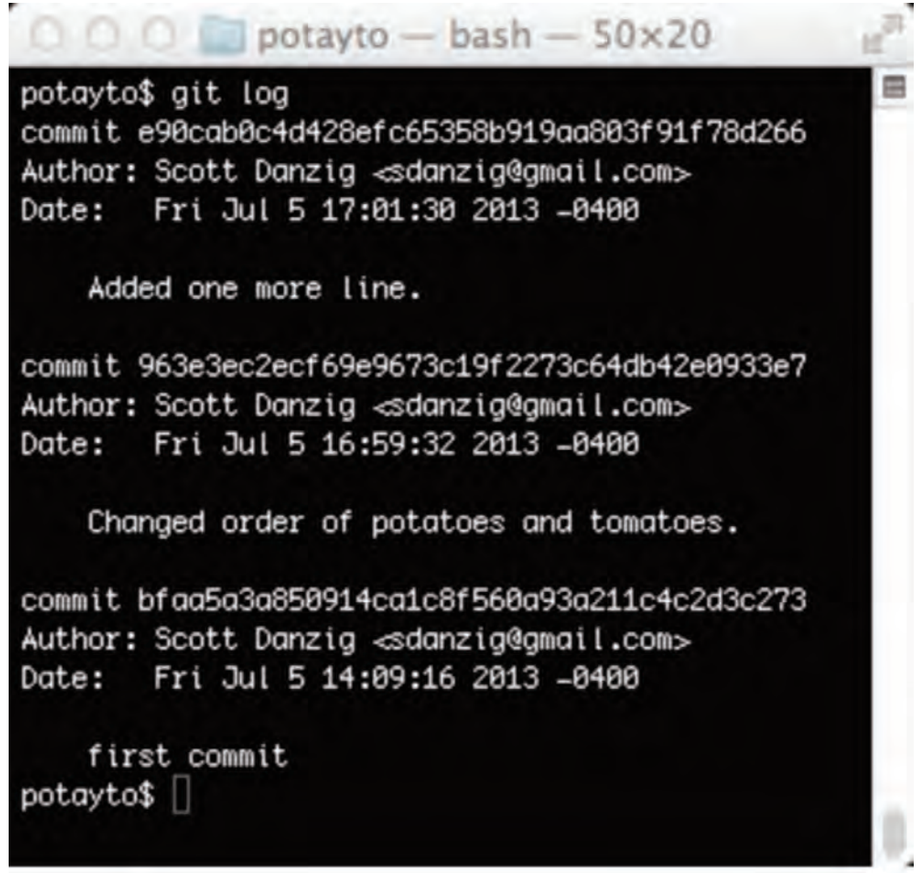


Figure 4: Output from git log.

Let's see how my first commit started. To see the details of the first commit, type: `git show bfaa`. Figure 5 shows the results.

At the bottom of Figure 5, you can see that I initially checked-in my Scala application as something that merely printed out "Tomayto tomahto," "Potayto potahto!" You can see that near the bottom. The `main()` method of the Potayto is executed, and there are those two print lines.

```
potayto — bash — 79×37
potayto$ git show bfaa
commit bfaa5a3a850914ca1c8f560a93a211c4c2d3c273
Author: Scott Danzig <sdanzig@gmail.com>
Date:   Fri Jul 5 14:09:16 2013 -0400

    first commit

diff --git a/.gitignore b/.gitignore
new file mode 100644
index 0000000..455ed5a
--- /dev/null
+++ b/.gitignore
@@ -0,0 +1,5 @@
+.cache
+.settings
+.classpath
+.project
+target
diff --git a/src/main/scala/scottdanzig/potayto/Potayto.scala b/src/main
scottdanzig/potayto/Potayto.scala
new file mode 100644
index 0000000..89c4043
--- /dev/null
+++ b/src/main/scala/scottdanzig/potayto/Potayto.scala
@@ -0,0 +1,10 @@
+package scottdanzig.potayto
+
+object Potayto {
+
+  def main(args: Array[String]): Unit = {
+    println("Tomayto tomahto,")
+    println("Potayto potahto!")
+  }
+
+}
\ No newline at end of file
potayto$
```

Figure 5: Contents of commit.

Earlier in Figure 5, you can see the addition of the `.gitignore` I provided. I'm making Git ignore my Eclipse-specific dot-something files (for example, Eclipse's .project) and also the target directory, where my source code is compiled to. Git's `show` command is showing the changes in this file, not the entire files. The `+`'s before each line mean the lines were added. In this case, they were added because the file was previously nonexistent. That's why you see the `/dev/null` there.

Now type `git show 963e` to get the output in Figure 6.

```
potayto — bash — 62×25
potayto$ git show 963e
commit 963e3ec2ecf69e9673c19f2273c64db42e0933e7
Author: Scott Danzig <sdanzig@gmail.com>
Date:   Fri Jul 5 16:59:32 2013 -0400

    Changed order of potatoes and tomatoes.

diff --git a/src/main/scala/scottdanzig/potayto/Potayto.scala
index 89c4043..132823c 100644
--- a/src/main/scala/scottdanzig/potayto/Potayto.scala
+++ b/src/main/scala/scottdanzig/potayto/Potayto.scala
@@ -3,8 +3,8 @@ package scottdanzig.potayto
 object Potayto {

   def main(args: Array[String]): Unit = {
-    println("Tomayto tomahto,")
-    println("Potayto potahto!")
+    println("Potayto potahto,")
+    println("Tomayto tomahto!")
   }

 }
\ No newline at end of file
potayto$
```

Figure 6: Commit message.

Here you see my informative commit message about what changed. These commit messages should be concise but comprehensive, so you're able to find the change when you need it.

After that, you see that I did exactly what the message says. I changed the order of the lyrics. You see two lines beginning with "-", preceding the lines removed; and two lines beginning with "+", preceding the lines added. You get the idea.

The .gitignore File, and Git Status

View the `.gitignore` file, which was dumped in Figure 5.

```
.cache
.settings
.classpath
.project
target
```

This is a manually created file in which I tell Git what to ignore. If you don't want files tracked, you add them here. I use the Eclipse IDE to write my code, and it creates hidden project files, which Git will see and want to add in to the project. Why should you be confined to using not only the same software as me to mess with my code, but also the same settings? Some teams might want to conform to the same development environments and checking-in the project files might be a time saver, but these days, there are tools that let you easily generate such project files for popular IDEs. Therefore, I have Git ignore all the Eclipse-specific files, which all happen to start with a "."

There's also a "target" folder in `.gitignore`. I've configured Eclipse to put my compiled code into that folder. We don't want Git tracking the files generated upon compilation. Let developers grabbing your source code compile it themselves after they make their modifications. You're going to want to create one for your own projects. This `.gitignore` file gets checked-in along with your project, so people who modify your code don't accidentally check-in their generated code as well. Other developers might be using IntelliJ IDE, which writes .idea folders and .ipr and .iws files, so they would add those to the `.gitignore` file.

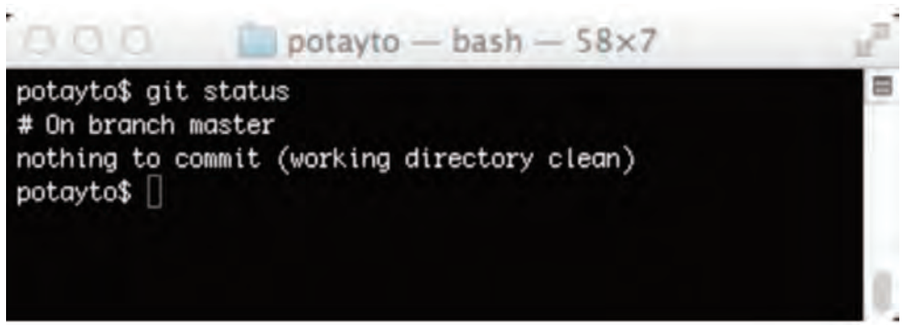


Figure 7: Status showing no new artifacts to commit.

Getting the Status

Now, let's try this. Type `git status`.

Figure 7 shows that there is nothing new to commit to your local repository. You also see in Figure 7 that you're on the main branch of your project, "master." Being "on a branch" means your commits are appended to that branch. Now create a text file named "deleteme.txt" using whatever editor you want in that potayto folder and type `git status` again, as in Figure 8.

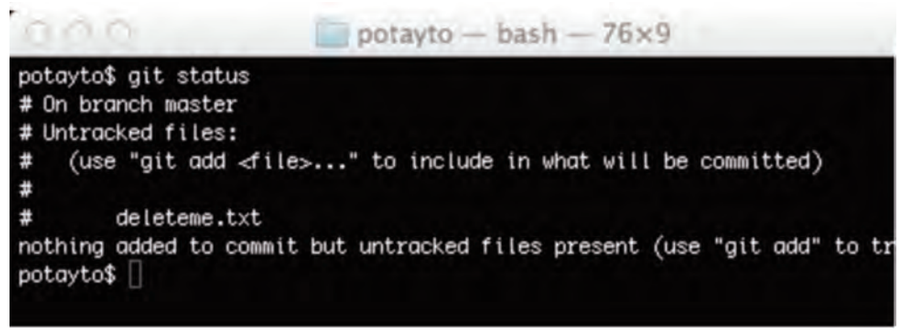


Figure 8: Status with artifacts to commit.

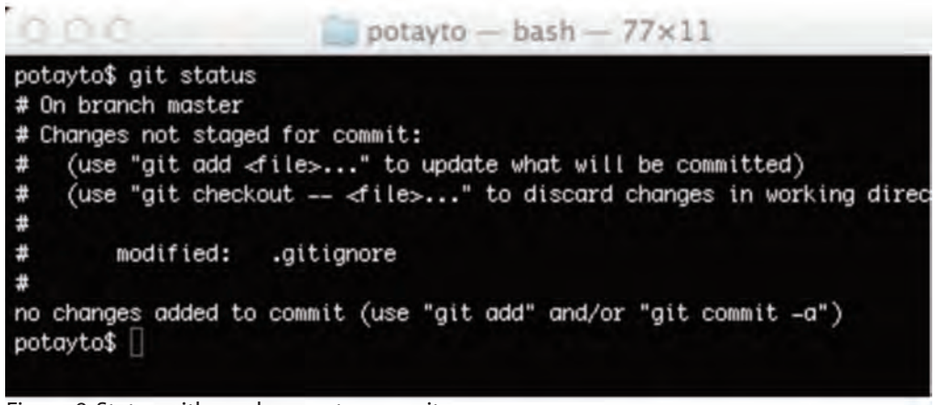


Figure 9: Status with no changes to commit.

Use that same text editor to add "deleteme.txt" as the last line of `.gitignore` and check this out (Figure 9).

Other than its special treatment by Git, `.gitignore` is a file just like any other file in your repository, so if you want the new information saved, you have to commit the change just like you would commit a change to your code.

Staging Changes

One of Git's best features is that it offers a staging process. You can stage the modified files that you want to commit. Other version control systems await your one command before your files are changed in the repository — generally the remote repository for the entire team. When you commit files in Git, files are held in a staging area. You will later commit all the files from the staging area to the larger repository.

So, let's say you wanted to make a change involving files `A` and `B`. You changed file `A`. You then remembered something unrelated to do with file `Z` and you modified that. Then you went back to your initial change, modifying file `B`. Git allows you to add files `A` and `B` to staging, while leaving file `Z` "unstaged." Then you can push only the staged files to your repository. But you don't! You realize you need to make a change to file `C` as well. You "add" it. Now files `A`, `B`, and `C` are staged, and `Z` is still unstaged. You commit the staged changes only.

Read that last paragraph repeatedly if you didn't follow it fully. It's important. See how Git lets you prepare your commit beforehand? With a version control system such as Subversion, you'd have to remember to make your change to file `Z` later, and your "commit history" would show that you changed files A and B, then, in another entry, that you changed file `C` later.

We won't be as intricate. Let's just stage our one file for now. Look at Figure 9. Git gives you instructions for what you can do while in the repository's current state. Git is not known for having intuitive commands, but it is known for helping you out. "`git checkout -- .gitignore`" to undo your change? It's strange, but at least it tells you exactly what to do. To promote `.gitignore` to "staged" status, type `git add .gitignore`.

The important thing to note in Figure 10 is that now your file change is listed under "Changes to be committed" and Git is spoon-feeding

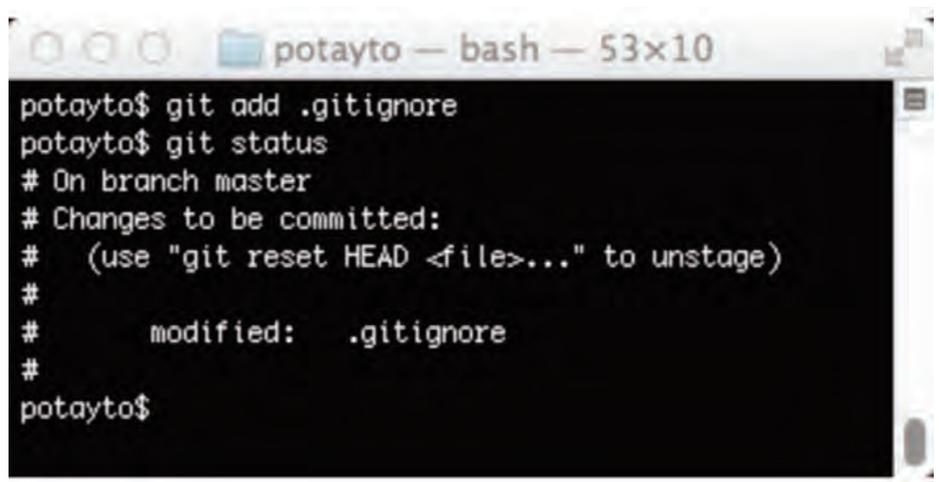


Figure 10: Promoting to staged status.

you what you need to type if you want to undo this staging. Don't type this: `git reset HEAD .gitignore`.

You should strive to understand what's going on (check out the Pro Git book I linked to for those details), but in this situation, you simply are given means to an end when you might need it (in case you change your mind about what to stage).

By the way, it's often more convenient to just type "`git add <folder name>`" to add all modifications of files in a folder (and subfolders of that folder). It is also very common to type the shortcut `git add .` to stage all the modified files in your repository. This is fine as long as you're certain that you're not accidentally adding a file such as `Z` that you don't want to be grouped into this change in your commit history. It's also useful to know how to stage the deletion of a file. Use `git rm <file>` for that.

Committing Changes to Your Repository

Time to do our first commit! To make the change in `.gitignore` official, type `git commit -m "Added deleteme.txt to .gitignore"` The `-m` option is followed by the commit message. You could just type `git commit`, but then Git would load up a text editor and you'd be required to type a commit message anyway. In Mac OS X and Linux, vim is the editor that would load up; and in Windows, you'd get an error. If you prefer a full screen editor in Windows, you can type this to configure it: `git config --global core.editor "notepad"`

If you end up in vim and are unfamiliar with it, note that it's a very geeky and unintuitive but powerful editor to use. In general, pressing the escape key, and typing ":x" will save what you're writing and then exit. The same syntax will work to choose a new full screen editor in OS X and Linux, of course replacing notepad with the `/full/path/and/filename` of a different editor.

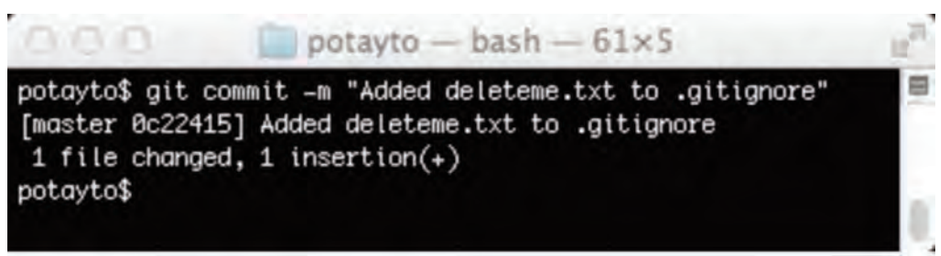


Figure 11: Committing with a commit message.

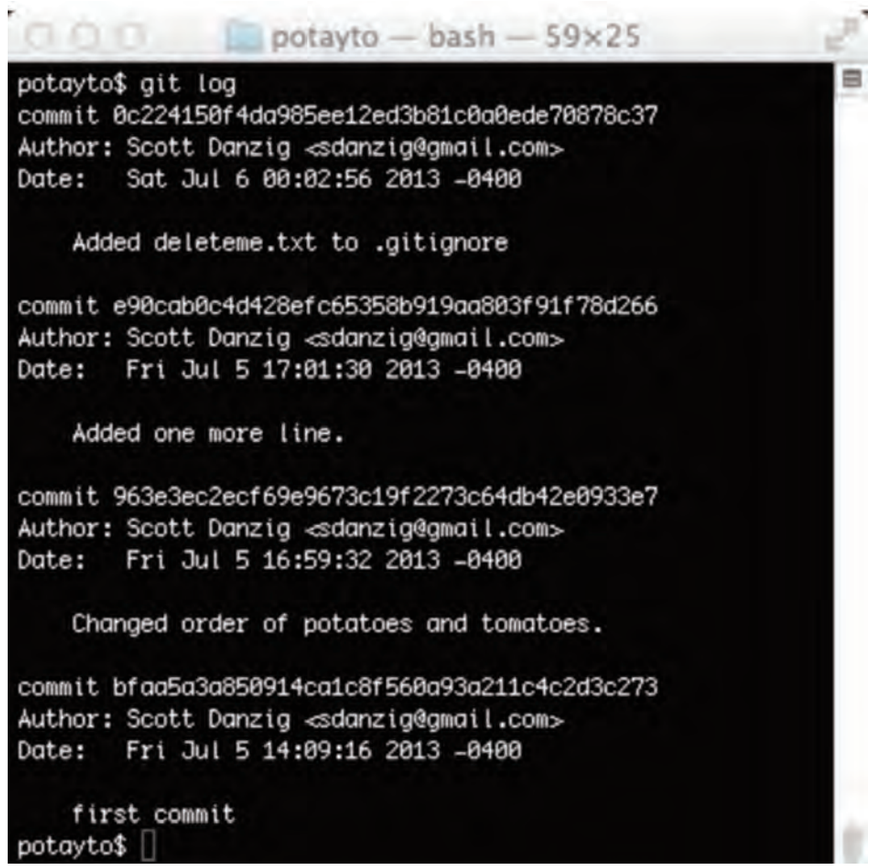


Figure 12: Log of the most recent commit.

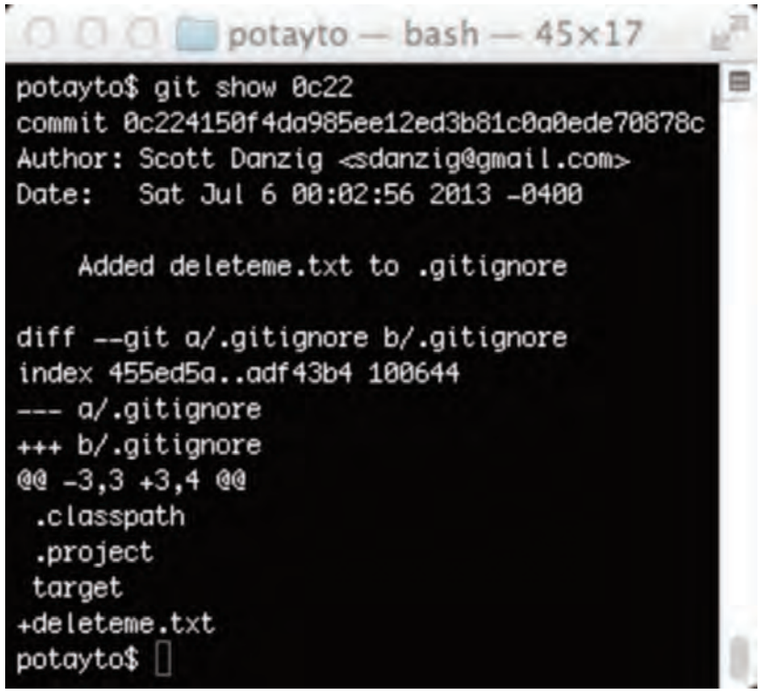


Figure 13: A diff output of the commit.

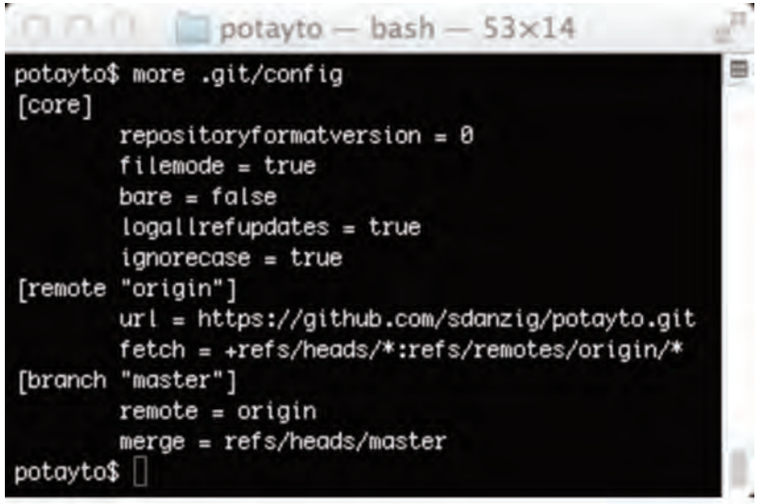


Figure 14: The contents of the .git/config file.

The full screen editor is necessary if you want a commit message with multiple lines, so if you hate vim, configure Git to use an editor you do like. To see the commit you just made, type `git log`.

The change on top is yours. Oh, what the heck, let's take a look at it with diff; see Figure 13.

The `+deleteme.txt` is the change that was just committed. The way this diff works is that Git tries to show you three lines before and after each of your changes. Here, there were no lines below your addition. The `-3,3` and `+3,4` are ranges. `-` precedes the old file's range, and `+` is for the new file. The first number in each range is a starting line number. The second number is the number of lines of the displayed sample before and after your modification. The 4 lines displayed only totaled 3 before your change.

Note that if you want to revert changes you made, the safest way is to use "git revert," which automatically creates a new commit that undoes the changes in another commit. If you wanted to undo that last commit, which has the SHA-1 starting with 0c22, you would type: `git revert 0c22`. (Don't actually do this if you are following along.)

Pushing Your Changes to Remote Repository

You cloned your repository from your GitHub account. Unless something went horribly wrong, the repository on GitHub should be: `https://GitHub.com/<your GitHub username>/potayto.git`

Git automatically labels the location you cloned a repository from as "origin." Remember when I said the internals of a Git repository were easily accessible in that .git folder in your project? Look at the text file .git/config; see Figure 14.

Before I explain how to make your changes on the version of your code stored on GitHub, I should first explain more about branches. I already noted how a branch is a separate version of your code. A change made to one branch does not affect the version of your repository represented by another branch, unless you explicitly merge the change into it. By default, Git will put your code on a "master" branch. When you clone a project from a remote repository ("remote" in this case means hosted by

GitHub), it will automatically create a local branch that "tracks" a remote branch. Tracking a branch means that Git will help you to:

- See the differences between commits made to the tracking branch (the local one) and the tracked branch (remote)
- Add your new local commits to the remote branch
- Put the new remote commits on your local branch

If you didn't have your local branch track the remote branch, you could still move changes from one to another, but it becomes more of a manual process. To do this, first, type `git status`.

That deleteme.txt change you made in your local master branch is not yet on Github! You have one commit that Github's (origin) remote master branch (denoted as origin/master) does not yet have, Figure 15.

Let's put the change on Github. Git's `push` command, if you don't provide arguments, will just push all the changes committed in your local branches to the remote branches they track. This can be dangerous, if you have commits in another local branch and you're not quite ready to push those out also. (I once accidentally erased a week of changes in *New York Magazine*'s main repository doing this. We did manage to recover them, but don't ask.) It's better to be explicit. Type `git push origin master`.

You don't really need to concern yourself with the details of how Git does the upload. But as for the command you just typed, Git's push lets you specify the "remote" that you're pushing to, as well as the branch (Figure 16). By specifying the branch, you tell Git to take that particular branch ("master," in this case) and update the remote branch, on the origin (your Github potayto repository) with the same name (it will create a new remote "master" branch if it doesn't exist). If you don't specify "master," Git will try to push the changes in all your branches to branches of the same names on the origin (if they exist there).

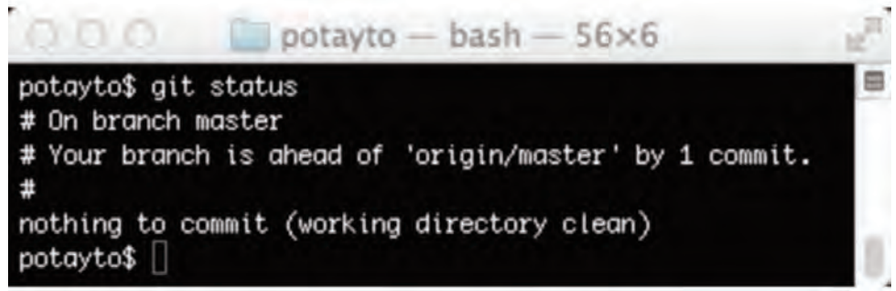


Figure 15: Showing that the local repository is ahead of the remote repository.

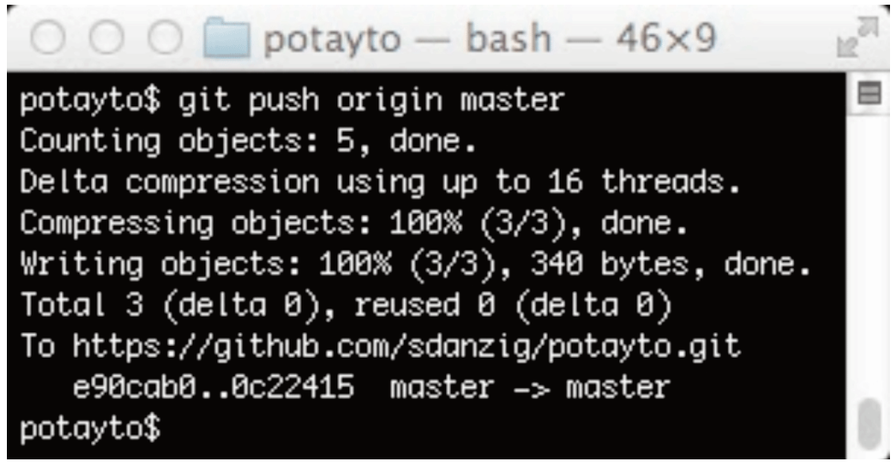


Figure 16: Pushing commit.

If you type "`git status`" again, you'll see your branch now matches the remote repository's copy of it. You can also look at the changes to the origin, by typing `git log origin/master`.

Figure 17 is the syntax to see a log of the commits in the master branch on your "origin" remote. You can see the change is there.

Figure 17: Log of changes to the origin.

You can also see this list of commits by logging into Github, viewing your Potayto repository, and clicking on the link shown in Figure 18.

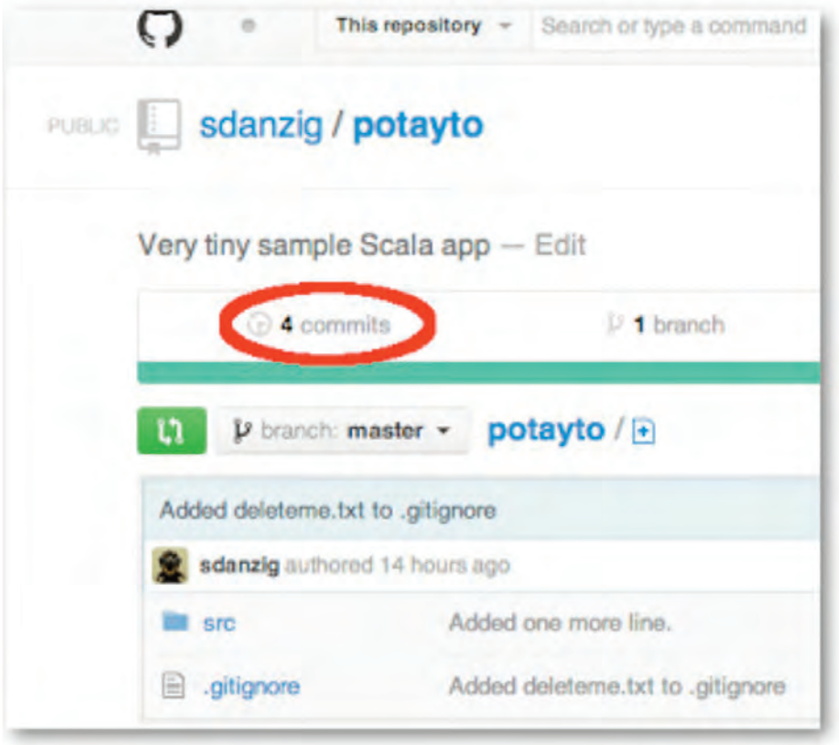


Figure 18: Seeing changes to the origin.

In the next installment of this tutorial, we'll examine how to pull changes from the remote repository, how to handle merges and merge conflicts, and other workflow tasks that are part of standard SCM work with Git.

— Scott Danzig has been programming for more than 20 years. His personal projects on Github can be found at https://Github.com/sdanzig.

Comment

Git Tutorial: Branches and Workflow

Pulling changes, handling merges and conflicts, and building a productive workflow are activities that Git handles in its own productive but unique way.

By Scott Danzig

This article is the second in a two-part tutorial on using Git. If you've never used Git, you should read the first installment *Getting Started with Git: The Fundamentals* on page 16 before starting on this one. In the previous article, I showed how to set up a Git project on GitHub, copy the project's files to a local repository, make changes locally, stage them, and finally push them to the remote repository.

Moving Forward

While we're browsing the Github interface, let's use it to create a change that you can fetch (or pull) to your local Git repository. This will emulate someone else accessing the remote repository and making a change. If you want your local copy of the repository to reflect what's stored in the remote repository, you need to keep yours up-to-date by intermittently fetching new changes. First, let's create a README.md file that Github will automatically use to describe your project. Github provides a button labeled "Add a README" for this, but let's do it the more generic way. Click the "Add a file" button on the GitHub repository (Figure 1).

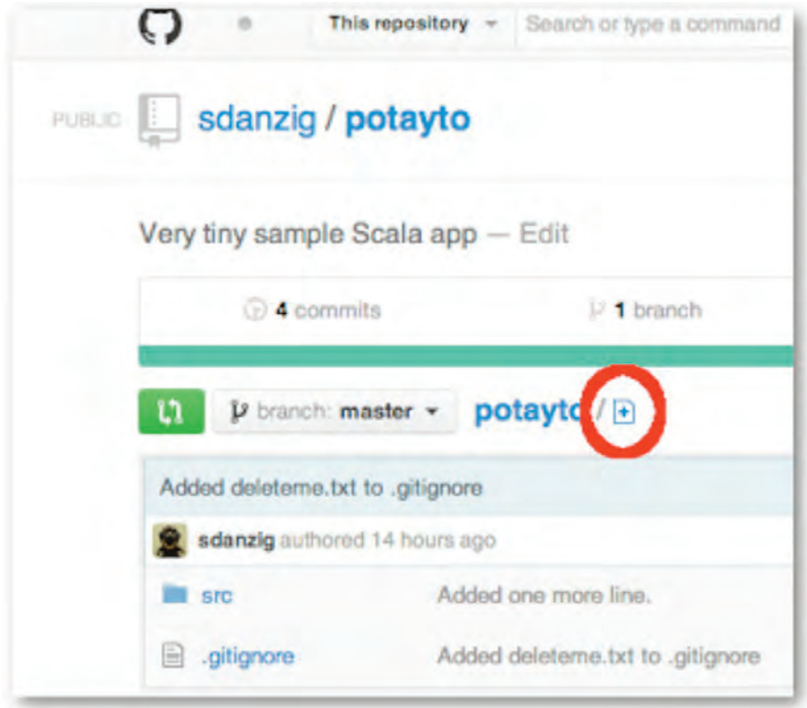


Figure 1: Adding a file to the remoted GitHub repository.

Type `README.md` for the name and a description that makes sense to you. (The "md" in the filename stands for "Markdown," which is a markup language that lets you augment your text with simple formatting. If you want details on how pretty you can make your README file, learn more about Github's version of Markdown (http://is.gd/QTxpRv). After adding some text, click the "Commit new file" button. You've committed the file to the remote GitHub repository. Go back to your terminal window and type `git status`.

In Figure 2, Git tells you that there's nothing to commit. This is because the Git `status` command does not do any network communication. Even typing "`git log origin/master`" won't show the change. Only Git's `push`, `pull`, and `fetch` do anything over the network. Let's talk about `fetch`, since `pull` is just a shortcut to some of the functionality that `fetch` offers.

When you track a remote branch, you do get a copy of that remote branch in your local repository. However, aside from those three aforementioned commands that talk over the network, Git treats these remote branches just like any other branches. (You can even have one local branch track another local branch.)

So, how do we update our local copies of the remote branches? `git fetch` will update all the local copies of the remote branches listed in your .git/config file. Figure 3 is what I get when I type `git fetch`.

Now, you'll notice there's still no difference if you type `git log`, but let's type `git log origin/master` (Figure 4), to see the remote change.

Let's now merge the change we made on the remote repository with our local repository. In Git, a clean merge like that is called "fast-forwarding." It means there's no potential conflict. Specifically, it means that no changes were made to the branch you're going to merge the changes into. I'll explain more later in the section on rebasing, but for now, we're going to pull these changes in to our local repository. Type: `git merge origin/master`.

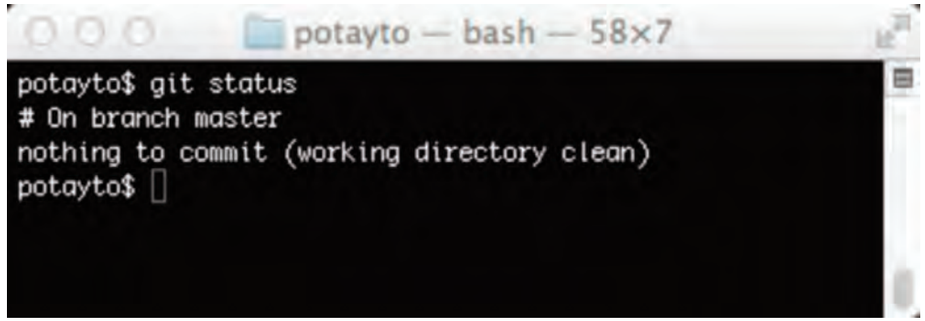


Figure 2: Status after a change to the remote directory.

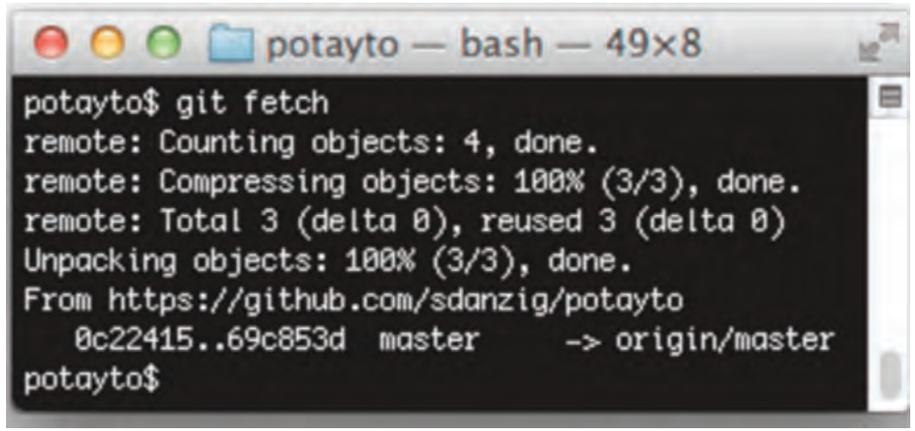


Figure 3: Output from git fetch.

Figure 5 shows there was one file inserted. Now if you typed `git log`, you'd see that you brought the change first from the master branch on your GitHub repository to your origin/master branch, and then from there to your local master branch. You could even have absolute proof of the change by looking in your current directory, where you'll see the README.md file.

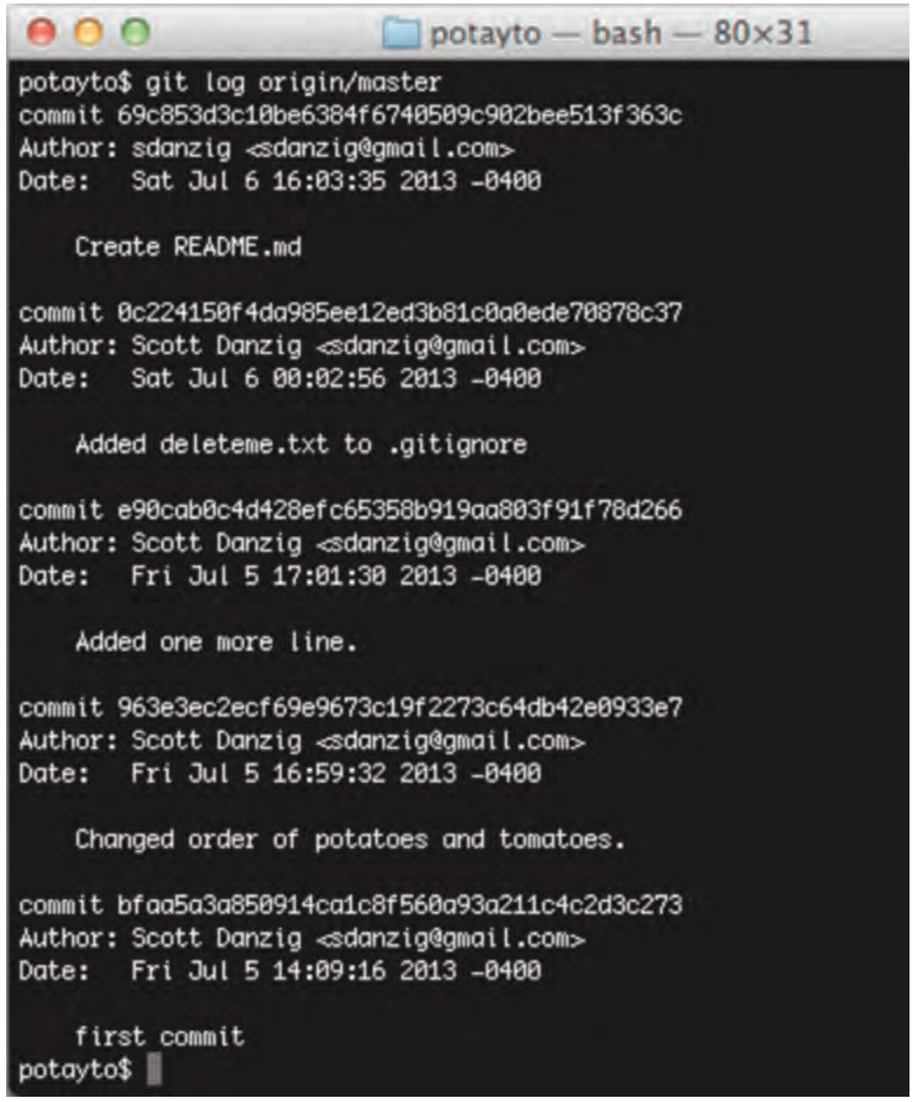


Figure 4: How changes appear in the master.

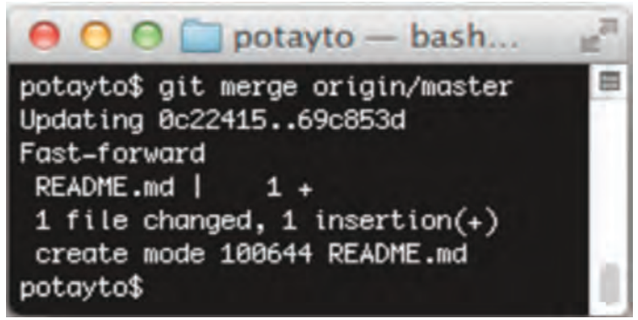


Figure 5: Pulling the remote changes from the master to our local branch.

There is a short cut. It's too late now that we've done the merge, but you could have done everything in one fell swoop by typing: `git pull origin master`.

That would have fetched the commits from the remote repository and done the merge. And if you want to pull all of the branches from all the remote repositories that your .git/config file lists, you can just type: `git pull`. You can be as trigger happy as you want.

Merges and Conflicts

For the purpose of learning about merges, we're going to undo that last merge. Very carefully, type `git reset HEAD~1 --hard`.

In Figure 6, "`HEAD~1`" refers to the first commit before the latest commit. The latest commit is referred to as the "`HEAD`" of the branch (currently master). By doing this hard reset, you're actually permanently erasing the last commit from your local master branch. As far as Git's

Figure 6: Undoing a merge.

concerned, the last link in the master branch's "chain" now is the commit that was previously second to last. Don't get in the habit of doing this. It's just for the purpose of this tutorial.

Your new README.md file is also safely committed to your local repository's cached version of the remote master branch, "origin/master." You could type `git merge origin/master` to remerge your changes, but don't do that right now.

Let's say someone else added that README.md, and you were unaware. You start to create a README.md in your local repository, with the intention of pushing it to the remote repository later. Because we undid our change, there is no longer a README.md file in your current directory.

Let's quickly create a new README.md file: `echo A test repository for learning git > README.md`.

I used the `cat` command (for Windows, it'd be `type`) to display the contents of the simple file we created to make sure it's right. Now, let's stage and commit it (Figure 7). Type:

```
git add README.md
git commit -m "Created a simple readme file"
git status
```

Note the file was created and that the divergence between repositories has been identified. At present, two versions of a README.md

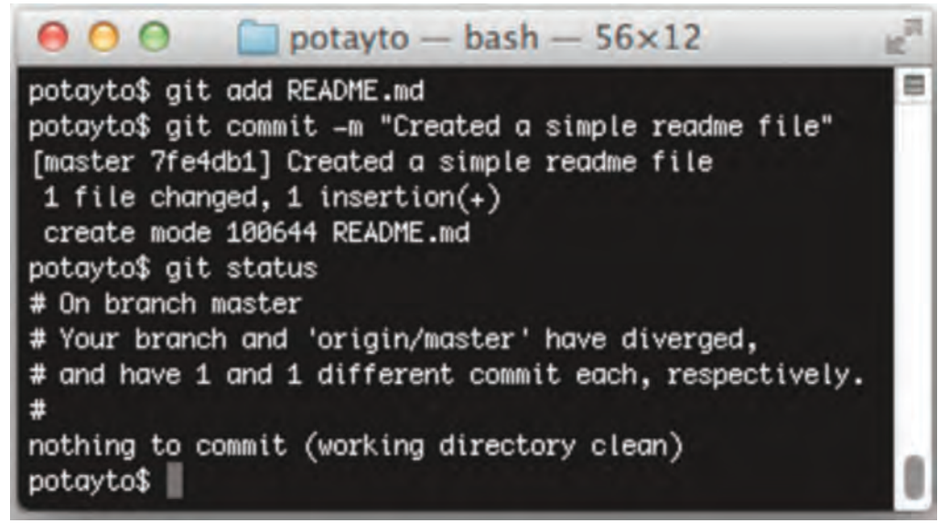


Figure 7: Stage and commit.

file committed. You can see that your origin/master branch is one commit in one direction, and your master branch is one commit in the other direction. What will happen when I try to update master from origin/master? Type `git merge origin/master` (Figure 8).

Just as you might think, Git is flummoxed. This is essentially Git saying "You fix it." Let's see what state we're in. Type `git status`.

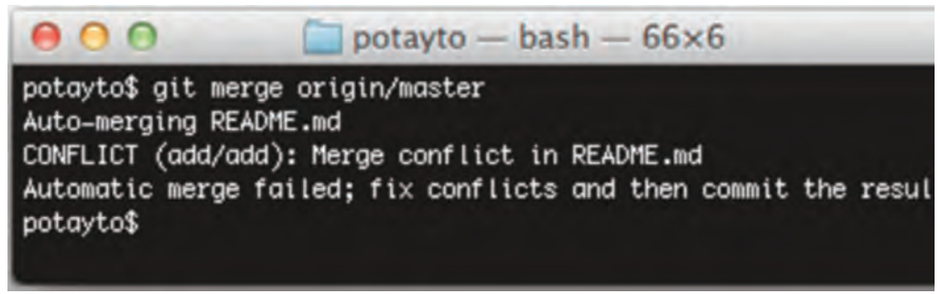


Figure 8: Error from a merge.

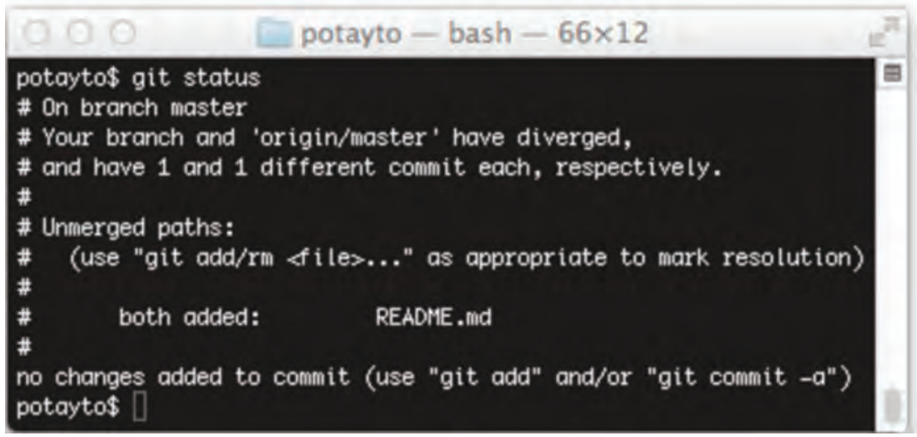


Figure 9: Status after a failed merge.

The message in Figure 9 can't be any clearer, except for one detail. You have two options at this point. You can either edit the local file to match the original, or you can have Git help you. Let's choose the latter path, which is what you'd always choose with complex conflicts. While still in your project directory, having just experienced a failed merge command, type `git mergetool`.

Mergetool will guide you through each conflicted file (Figure 10), letting you choose which version of each conflicted line you'd like to use for the committed file. You can see, by default, it uses opendiff. Press enter to see what opendiff looks like (Figure 11).

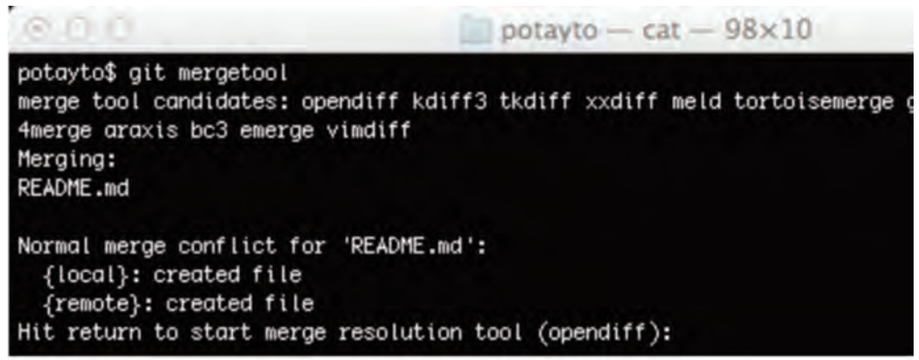


Figure 10: Output from running git mergetool.

If this were a conflict of more than one line, you'd be able to say "use the left (or right) version for this conflict line," or even "I don't want to use either line." In this case, we only have one conflicted line to choose from. Click on the "Actions" pull down menu and choose "Choose right." You'll see nothing has changed. That's because that arrow in the middle was already pointing to the right. Try selecting "Choose left," then "Choose right" again. You'll see what I mean. Opendiff doesn't give you the opportunity to put in your own custom line. You can do that later if you wish. At the pull down menu at the top of the screen, select "File" then "Save Merge," go back to the menu and select "Quit FileMerge." Now, to stage the new version of the README file. Type `git add README.md`.

Now you're all set to commit changes, just like if you manually modified and staged (with git add) the files yourself. Now type `git commit -m "Merged remote version of readme with local version."` and then `git status`.

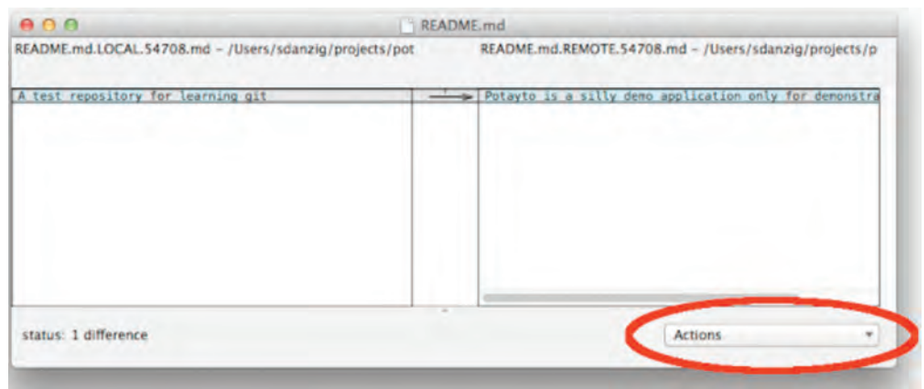


Figure 11: Opendiff.

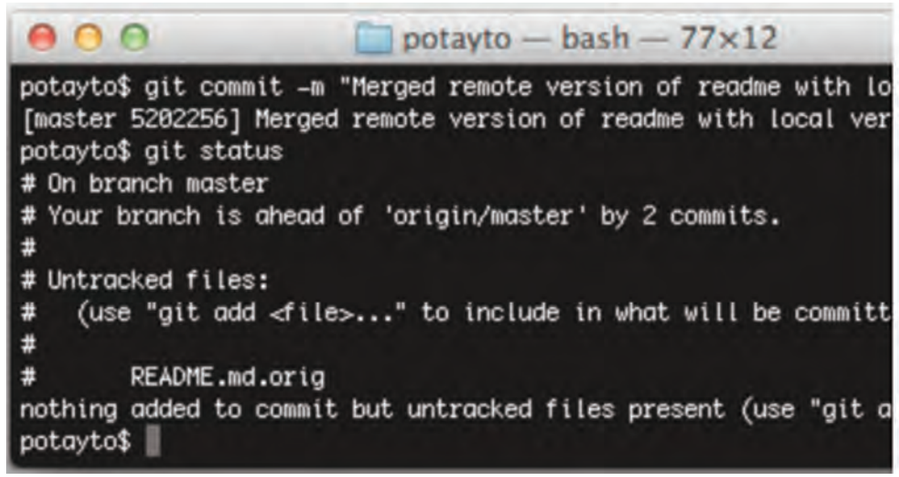


Figure 12: Status after the new commit.

Before we go on, if you noticed, there's a lingering "README.md.orig" file (Figure 12). That is just a backup file. However, it's a pain to deal with these "orig" files. For this time, you can move the file somewhere else, or just delete it, but for future reference, check out http://is.gd/rABRwx, a page on the many strategies you can leverage to deal with those files.

Back to the merge. Look! Your branch is "ahead" of "origin/master" by 2 commits. Let's see what those commits are. To show just the last two commits, type `git log -n 2`.

Now, let's push our changes to origin/master and see what happens. Type git push origin master. Now, just to be sure, we're not going to look at the "local version" of the remote branch. Let's go right to Github to see what happened. View the commits in your repository.

What might not make sense here, is that you have first the GitHub-side readme commit, then your local readme commit, then the merge. It doesn't make sense for all of these commits to happen in sequence,

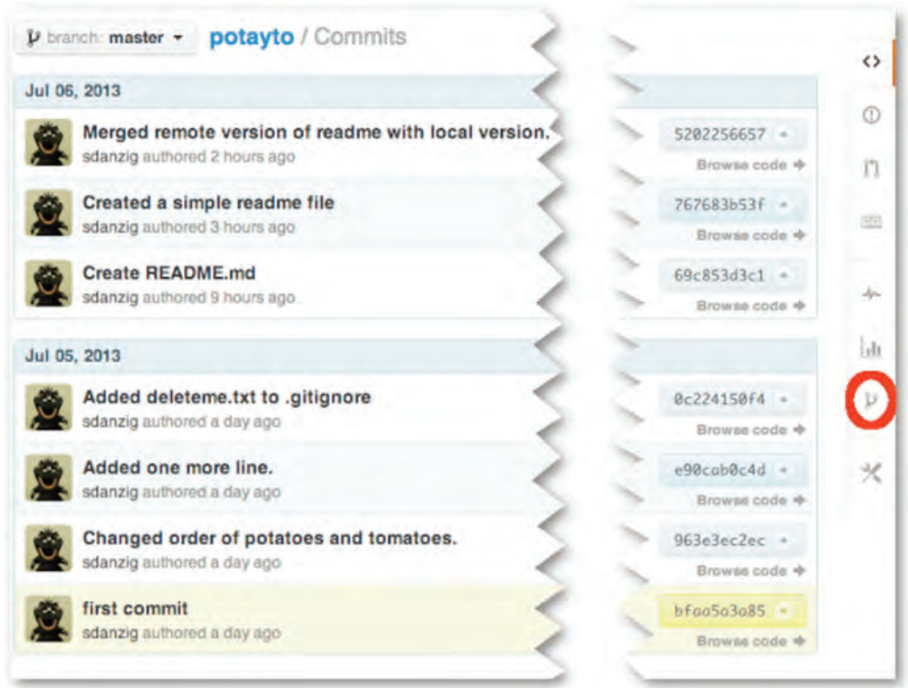


Figure 13: GitHub page showing commits.

since the first two are conflicting. What happens is that your local readme file commit is logged as a commit on a separate branch that is merged in. Let's graphically demonstrate that by clicking on the "Network" button on the right (circled in red in Figure 13).

Each dot in Figure 14 represents a commit. Later commits are on the right. The one that looks like it was committed to a separate branch (your local master branch) and then merged in is the commit of your local version of the readme file. Hover over this dot and see for yourself.

Rebasing

Before heading into discussions of workflows, I want to touch on a fea-

Figure 14: GitHub's timeline on commits and merges.

ture that Git does uniquely well, and that's worth knowing about should the need ever arise. It's called "rebasing." With it, you can shape your commits the way you prefer before merging them to another branch. You can already do some preparation when you're staging your files. You can stage and unstage files repeatedly, getting a commit exactly how you want. But there are two main things that rebasing lets you do in addition to that.

Let's say you were working on branch `A` and you created branch `B`. Branch `B` is nothing more than a series of changes made to a specific version of branch `A`, starting with a specific commit in branch `A`. Let's say you were able to take those changes and reapply them to the last commit in branch `A`! It's as though you checked out branch `A` and you made the same changes. You can use rebasing to allow your merges to be "fast-forward," so when you merge subsequent changes into another branch, there's no "merge commit." Your changes are simply added as the next commits in the target branch, and the new latest commit of that branch is your last change. This is a powerful feature.

Git Workflows

One of the most common Git workflows is the pull request, which shows up a lot in open source projects. Commits are often grouped into "feature branches," representing all the changes needed for a branch. Projects with designated maintainer(s) often operate is as follows:

- You initially push your "feature branch" to a remote repository. This is often your fork of the main repository.
- You create a "pull request" on Github for that branch, which tells the project maintainer that you want your branch merged into the master branch.
- If the branch is recent enough or it can be rebased onto master without any conflicts, the maintainer can easily merge in your changes.
- If there are conflicts, then it's generally up to the maintainer to do the merge or to reject the pull request and let you rebase and "de-conflict" the commits in your branch yourself.

My Workflows

At *New York Magazine,* where I work, we generally have four main branches of each project entitled `dev`, `qa`, `stg`, `prod`.

- `dev` branch: While developers first test their code on their own computers, eventually they need to test changes on a server with shared resources. This exposes a bunch of integration issues and often requires multiple commits (multiple attempts to get it right) before the change is complete.
- `qa` branch: This is branch is for QA (quality assurance) testing to be done on a new change. The branch is cleaner, consisting only of completed changes. While everything isn't necessarily optimized (maybe you do have debugging information being recorded to the log, for instance), it's much more controlled as opposed to `dev`.

- `stg` branch: Changes approved by QA go to the "staging" environment. This environment is fully optimized, as if it were the production environment. There could be more issues that are exposed by testing in a fully optimized environment, but usually not. This is not to be confused with the much lower-level staging in Git, but ultimately, the concept is the same. You're ultimately preparing a set of features that are slated to go public, rather than a bunch of file changes that are about to be committed.
- `prod` branch: What your clients/customers/users ultimately see is deployed directly from this branch.

We rely on the open-source continuous integration server Jenkins (http://jenkins-ci.org/) to monitor each branch. When any change is made, the project is built and redeployed to a computer/server dedicated to that environment. To manage the environment-specific configuration, including enabling optimizations and altering logging levels, we use Puppet (http://puppetlabs.com/puppet/). We also use Git to maintain our internal documentation, written as text files using the Git-variety of Markdown, to allow ease of collaboration and code-friendly formatting.

Each commit message at the magazine, optimally, should have a story number. A "story" is a description of a desired modification. If something should be changed in code, someone describes how the change works in a web interface provided by a story-tracking application such as Atlassian's JIRA (https://www.atlassian.com/software/jira), which we use. A developer can modify the "status" of the story to reflect progress being made toward its resolution.

We use Crucible (https://www.atlassian.com/software/crucible) for peer code reviews. This lets a developer send a series of commits out to fellow developers to have a look at. It tracks who has made a change to review your code, and gives them the opportunity to make comments.

I'm often tasked with a modification I must make to a shared project hosted as a Github repository as I described. On Github, I have a separate user, "scottdanzig" for my Github activity, which allows clear separation of my personal projects from what I've done that for the magazine. For my examples, I'll refer to a Web application created with Scala and the Play Framework, that provides restaurant listings for your mobile device. Let's say we realized that the listings load very fast, and we can afford to display larger pictures. Here is my preferred workflow:

- The first thing I do is change the status of the JIRA story I'm going to work on to "In Progress."
- If I don't yet have the project cloned onto my machine, I'll do that first: `git clone https://GitHub.com/nymag/listings.git`
- Check out the `dev` branch: `git checkout dev`
- Update my `dev` branch with the latest from the remote repository: `git pull origin dev`
- Create and checkout a branch off `dev`: `git checkout -b larger-pics`
- Make my modifications and test as much as I can, staging and committing my changes after successfully testing each piece of the new functionality.
- Then update my `dev` branch again, so when I merge back, hopefully it's a fast-forward merge: `git pull origin dev`
- I'll interactively rebase my `larger-pics` branch onto my `dev` branch. This gives me an opportunity to change all my commits to one big commit, to be applied to the latest commit on the `dev` branch: `git rebase -i dev`. I write one comprehensive commit message detailing my changes so far, making sure to start with the JIRA story number so people can review the motivation behind the change. It's possible I might want to not com-

bine all my commits yet. If I'm not sure if one of the incremental changes is necessary, I may decide to keep it as a separate commit. This is possible if you leave it as a separate "pick" during the interactive rebasing. Git will give you an opportunity to rewrite the commit description for that commit separately.

- Checkout the `dev` branch: `git checkout dev`
- Merge in my one commit: `git merge larger-pics`
- Push it to Github: `git push origin dev`
- If Git rejects my change, I may need to rebase my `dev` branch onto `origin/dev`, and then try again. We're not going to combine any commits, so it doesn't need to be interactive: `git rebase origin/dev` then again: `git push origin dev`
- Jenkins will detect the commit and kick off a new build. I can log into the Jenkins Web interface and watch the progress of the build. It's possible the build will fail, and other developers will grumble at me until I fix the now broken dev environment. Let's say I did just that.
- If I think it might be a while before I'm able to fix my change, I'll use "`git revert <SHA code>`" to undo the commit. Either way, I'll again checkout my `larger-pics` branch, `git rebase dev`, then make changes, `git pull origin dev, git rebase dev, git checkout dev, git merge larger-pics, git push origin dev`. Let's say Jenkins gives me the thumbs up now.
- The next stage is the code review. I'll log into Crucible and advertise my list of commits in the `dev` branch for others to review. I can make modifications based on their feedback if necessary.
- Say both Jenkins and my fellow developers are happy. It's time to submit my code to QA. The `qa` branch is automatically deployed by Jenkins to the QA servers, a pristine environment meant to better reflect what actually is accessed by *New York Magazine*'s readers. We have some dedicated QA experts who systematically test my functionality to make sure I didn't unintentionally break something. If there are no QA experts available, QA might be done by another developer if the feature is sufficiently urgent.
- I need to update my local QA branch so I can rebase my changes onto it, pushing fast-forward commits. I first type: `git pull origin qa`
- Then I change to my `larger-pics` branch: `git checkout larger-pics`
- It's time to rebase my commits onto the `qa` branch, rather than `dev`, which can be polluted by the works in progress of other developers. I type: `git rebase -i qa`, creating a combined commit message describing my entire set of changes. I now have a branch that is the same as QA, plus one commit that reflects all of my changes.
- Add my branch to the remote repository: `git push -u origin larger-pics`
- I go to the repository on Github and create a pull request (see http://is.gd/nLaVwH), requesting my `larger-pics` branch be merged into the `qa` branch.
- At this point, it's out of my hands for the time being. However, the project has a "maintainer" assigned.
- The maintainer can first use the Github interface to see the changes (http://is.gd/W0hWd6). The maintainer can give a last check for the code.
- If approved, the maintainer must merge the branch targeted by the pull request to the `qa` branch. If the commit will have no conflicts, Github's interface is sufficient to merge the change. Otherwise, the maintainer can reject the change, requesting for the

original developer of the change to rebase the branch again and resolve the conflict before creating a new pull request. Otherwise, the maintainer can check out the branch locally and resolve the merge, rather than the original developer doing it.
- The maintainer commits the merged change and updates the JIRA story to "Submitted to QA."
- If QA finds a bug, they will change the JIRA status to "Failed QA." The maintainer will checkout the QA branch and use "`git revert`" to roll back the change, then will reassign the JIRA ticket back to the original developer.
- If QA approves the change, however, they will change the JIRA status to "Passed QA."

At regular intervals, a development team will release a set of features that are ready. A release consists of:

- A developer merging QA-approved changes from the QA branch to the staging branch.
- Members of the team having a last look at the change's functionality in the staging environment.
- The developer of a change, after confirming that it works correctly in staging, merges the change into the `prod` branch before a designated release cutoff time.
- The developer changes the status of the JIRA story to "Resolved"
- The system administrators deploy a build including the last commit before the cutoff time. For us, this entails a brief period of down-time, so the release is coordinated with the editors and others who potentially will be affected.

Further Thoughts

That's a summary of how I work, and although everything is sensible, it's a bit in flux. These are things which could be changed:

- We can get rid of the staging environment, and merge directly from QA. I see the value in this extra level of testing, but I believe four stages is a bit cumbersome.
- A project does not necessarily need a maintainer, and if we use Crucible, perhaps not even pull requests. A developer can merge his change directly into the QA branch and submit the story to QA on his/her own. I prefer to have a project maintainer.
- We can get rid of Crucible, and just use the code review system in Github. It might not be as feature-filled, but if we use pull requests, it's readily available and could streamline the process. I like Crucible, although it might be worth exploring eliminating this redundancy.

After years of using many other version control systems, Git has proven to be the one that makes the most sense. It's certainly not dependent on a reliable Internet connection. It's fast. It's very flexible. After more than twenty years of professional software development, I conclude Git is an absolutely indispensable tool.

— Scott Danzig has been programming for more than 20 years. His personal projects on Github can be found at https://Github.com/sdanzig.

Comment

This Month on DrDobbs.com

Items of special interest posted on www.drdobbs.com over the past month that you may have missed

THE EVENTUAL EVOLUTION OF BUILD SYSTEMS

Build systems are deeply siloed by development language and primarily rely on the '80s model of spelling out targets and instructions. But new tools, such as Gradle, that cater to polyglot apps and the needs of continuous delivery are finally emerging.

http://www.drdobbs.com/240160138

AN IMPORTANT MOVE OPTIMIZATION IS NEARLY INVISIBLE

Using a `list` instead of a `vector` carries its own overhead. Although there is no need to move elements in order to expand a `list`, it is necessary to allocate dynamic memory for each element as part of creating the element. Moreover, because these elements are dynamically allocated, they are unlikely to be in contiguous memory, a fact that makes it less likely that the processor will be able to use its hardware cache to make accesses to those elements more efficient.

http://www.drdobbs.com/240160031

INTERNATIONALIZATION: FROM THE SUBLIME TO THE RIDICULOUS

Making apps available in foreign languages can be difficult and filled with so many unexpected and illogical barriers that developers are motivated to pay little attention to users outside their own cultures.

http://www.drdobbs.com/240159532

THE PI TRANSFORM

When I started this series of blogs, I asked: "Can you do signal processing on the Pi?" I think the answer is a resounding yes.

http://www.drdobbs.com/240159448

WORKING WITH PROTOBUF WCF SERVICES

Protocol Buffers are a serialization format, developed by Google, that is fast and easy to use. Frequently referred to as "Protobuf," the technology consists of a language- and platform-neutral, extensible, binary format for fast data exchange that relies on an interface description language, which is compiled to native code.

http://www.drdobbs.com/240159282

IN THIS ISSUE

Dr. Dobb's

Andrew Binstock Editor in Chief, Dr. Dobb's
andrew.binstock@ubm.com

Deirdre Blake Managing Editor, Dr. Dobb's
deirdre.blake@ubm.com

Amy Stephens Copyeditor, Dr. Dobb's
amy.stephens@ubm.com

Jon Erickson Editor in Chief Emeritus, Dr. Dobb's

DR. DOBB'S EDITORIAL
751 Laurel Street #614
San Carlos, CA
94070
USA

UBM TECH
303 Second Street,
Suite 900, South Tower
San Francisco, CA 94107
1-415-947-6000